

N° d' enregistrement
au C.N.R.S.

THESE DE DOCTEUR INGENIEUR

Mathématiques Appliquées

présentée

A L' UNIVERSITE PIERRE ET MARIE CURIE (PARIS VI)

par

M. Jacques FERBER

pour obtenir

LE TITRE DE DOCTEUR INGENIEUR

Sujet de la Thèse : MERING: un langage d'acteur pour la
représentation et la manipulation
des connaissances.

Soutenue le 20 décembre 1983

devant la commission d'Examen

MM. J.C. SIMON Président

P. GREUSSAY Examineurs

Y. HAMAN

D. KAYSER

J.F. PERROT

G. SABAH

THESE DE DOCTEUR INGENIEUR

Mathématiques Appliquées

présentée

A L'UNIVERSITE PIERRE ET MARIE CURIE (PARIS VI)

par

M. Jacques FERBER

pour obtenir

LE TITRE DE DOCTEUR INGENIEUR

Sujet de la Thèse : MERING: un langage d'acteur pour la
représentation et la manipulation
des connaissances.

Soutenue le 20 décembre 1983

devant la commission d'Examen

MM. J.C. SIMON Président

P. GREUSSAY Examineurs

Y. HAMAN

D. KAYSER

J.F. PERROT

G. SABAH

Je remercie monsieur le professeur Jean-Claude SIMON et lui suis très reconnaissant d'avoir accepté de présider le jury.

Je remercie tout particulièrement monsieur Gérard SABAH pour avoir su guider mes travaux de son amitié exigeante et de ses conseils. Je tiens à le remercier également d'avoir accepté d'être le rapporteur de cette thèse.

Je tiens à remercier monsieur le professeur Patrick GREUSSAY, en qui j'ai une grande admiration, d'avoir accepté d'être examinateur dans ce jury.

Que le professeur Jean François PERROT trouve ici l'expression de ma gratitude pour son accueil chaleureux lorsque je me suis présenté à lui, pour ses remarques et ses conseils qui m'ont été très utiles, et aussi pour avoir accepté d'être examinateur.

Je remercie également monsieur le professeur Daniel KAYSER d'avoir accepté d'être examinateur dans ce jury. Je lui suis très reconnaissant de l'intérêt qu'il a manifesté pour mes travaux.

Je remercie le professeur Y. HAMAM d'avoir bien voulu accepter d'être examinateur dans ce jury.

Je tiens à associer à ces remerciements Jean Baptiste Berthelin, Brigitte Grau, Mohamed Rady, Pierre Cointe, Jean Sallantin et Harald Wertz qui m'ont soutenu par leur amitié et leurs conseils.

Par dessus tout, je tiens à te remercier Annick "What we cannot express in words, we must therefore pass over in silence" (Wittgenstein, Tractatus Logico-Philosophicus, §7).

TABLE DES MATIERES

Introduction	1
 Chapitre I Le formalisme objet dans la représentation des connaissances	 6
1 Généralités et historique	6
2 Le niveau conceptuel	10
3 Formalisme objet et relationnel	14
4 Sept critères pour caractériser un langage de représentation et de manipulation de connaissances	21
5 Les réseaux sémantiques	25
6 Les schémas	27
7 Les langages orientés objet	29
8 Conclusion	31
 Chapitre II Le langage MERING	 32
1 Les concepts de base	32
1.1 Un langage d'acteur	33
1.2 Un langage de programmation pour la représentation des connaissances	37
2 Description de MERING	40
2.1 Présentation générale	40
2.2 Mering en 13 points	43
2.3 La syntaxe de MERING	46

3	Les transmissions	48
3.1	A-transmissions et M-transmissions	48
3.2	Cascades et enchaînements	51
3.3	Les fonctionnelles	52
3.3.1	Fonctionnelles de contrôle	53
3.3.2	Fonctionnelles de quantification	55
3.3.3	Fonctionnelles d'évaluation	57
3.3.4	Fonctionnelles d'entrées-sorties	60
3.4	Les continuations	60
4	Les entités	63
4.1	Types et représentants	64
4.2	Héritage des propriétés	66
4.3	Les comportements	71
4.4	Les attributs	75
4.4.1	Modèle général	75
4.4.2	Les réflexes: des mécanismes d'inférence	78
4.4.3	Les facettes	82
4.5	Définition des entités	91
4.5.1	Construction des types et des représentants	91
4.5.2	Description des méthodes	93
4.5.3	Description des applicateurs	94

Chapitre III Applications et extensions 96

1	Illustration sommaire: une base de données	97
2	Questionnaire intelligent: une réservation de place de chemin de fer	104
3	Un jeu d'aventure	117
4	Programmation par contraintes: application aux circuits électriques	131
4.1	La constitution des composants de base	132
4.2	La composition des circuits	135
4.3	La définition de types de circuits	138
5	Types et Classes	143
5.1	Généralités	143
5.2	Les classes en MERING	147
5.3	L'union des classes	152
5.4	L'intersection de classes	155
6	La programmation par règles de production	158
6.1	Inférences et règles de production	158
6.2	Utiliser des règles de production en MERING	162

6.3	Définition d'une règle	167
6.4	Intérêt des règles de production	169
6.5	Les paquets de règles	174
6.6	Description d'un analyseur de langage naturel en MERING ..	180

Chapitre IV Implantation et manuel de référence du langage 187

1	Structure de l'interpréteur MERING	188
1.1	L'évaluateur	188
1.2	La structure des entités	191
1.3	Boucle externe et continuation	196
1.4	Les fonctionnelles	198
2	Manuel de référence	200
2.1	Les entités de base	200
2.2	Description des comportements de base	206
2.2.1	Méthodes générales	207
2.2.2	Tests de types	211
2.2.3	Manipulation de facettes	215
2.2.4	Applicateurs actifs	219
2.2.4.1	La lecture d'attributs	220
2.2.4.2	Affectation et ajout de valeurs	223
2.2.4.3	Suppression de valeurs	228
2.2.5	Prédicats de correspondance	229
2.2.5.1	Les prédicats d'égalité	230
2.2.5.2	Les prédicats d'appartenance	233
2.2.5.3	Les prédicats de filtrage	235
2.2.6	Les manipulateurs de listes	240

CONCLUSION 247

ANNEXES 250

REFERENCES BIBLIOGRAPHIQUES..... 260

INTRODUCTION

La représentation des connaissances est une des branches de l'Intelligence Artificielle qui connaît à l'heure actuelle de grands développements.

Ce domaine pose deux questions essentielles:

- 1 - Quelles sont les catégories conceptuelles nécessaires à l'établissement d'une base de connaissance.
- 2 - Quelles sont les formulations informatiques les plus adéquates pour représenter ces catégories conceptuelles.

La première question fait l'objet d'une réflexion sur la conceptualisation, en distinguant les différentes entités susceptibles d'intervenir dans notre connaissance du monde et les liens statiques et dynamiques qui associent ces concepts entre eux ou qui permettent d'en déduire de nouveaux (objets, propriétés, relations, états, actions, événements, scénario, inférences, méta-connaissances, etc..) et sur l'utilisation qui est faite de cette connaissance: traitement (ce que l'on dénomme souvent sous le terme général de raisonnement), accès aux informations (ou processus de remémoration) et acquisition de nouvelles données (les mécanismes d'apprentissage).

La deuxième question, et c'est celle qui nous intéresse ici, concerne les moyens informatiques mis en oeuvre pour représenter et manipuler ces concepts.

Les techniques de représentation employées s'articulent autour de l'opposition entre "formalisme objet" et "formalisme relationnel ou prédicatif".

Dans le **formalisme relationnel** un objet n'existe pas en tant que tel, mais seulement comme participant à un ensemble d'énoncés dispersés dans la base. Le travail de raisonnement consiste alors à manipuler ces énoncés afin de déduire d'autres énoncés. Dans ce cadre, il n'est jamais question d'objets, mais de ce qui peut être dit sur ces objets: les langages PLANNER et PROLOG, se rangent dans cette catégorie.

A l'opposé dans le **formalisme objet**, les entités ont une existence réelle, un espace propre dans la base de connaissance. La représentation est alors une simulation, une image des choses et des phénomènes. Il ne s'agit plus ici d'affirmer quelque chose, mais de modéliser ce qui est ou ce qui se passe.

Les modèles conçus autour du formalisme "objet" ont conduit:

- 1) à l'élaboration de la notion de schéma (ou 'frame' en anglais). Technique de représentation issue de la généralisation des listes de propriétés, un schéma est défini comme une liste d'attributs (ou slots), chaque attribut étant lui même une généralisation du couple <attribut,valeur>, pour y incorporer des caractéristiques (hiérarchie des concepts, valeurs par défaut, contraintes, démons, etc..)
- 2) à la réalisation de langages pour manipuler et traiter ces schémas tels que FRL, KRL, AIMDS ou UNITS,

D'autre part, Les langages orientés objets tels que SIMULA, SMALLTALK, OBJVLISP et GLISP proposent un formalisme qui s'apparente aux schémas et permet donc de représenter, et surtout de manipuler des connaissances, grâce aux éléments suivants:

- Structuration de la base par différenciation entre classes et représentants

- Champs locaux pouvant s'apparenter aux attributs

- Intégration des caractères liés à la représentation et au traitement des connaissances grâce à l'emploi des méthodes.

Cependant il leur manque un certain nombre des caractéristiques des schémas (valeurs par défaut, démons, contraintes), ce qui les rend peu adaptés à la représentation des connaissances dans des domaines à fortes tendances inférentielles, tels que la compréhension d'histoire, le dialogue en langage naturel, les systèmes experts, etc...

Pour répondre à cette imperfection, nous avons élaboré un langage "orienté objet" destiné à la fois à la représentation et à la manipulation des connaissances: MERING (METHode de Représentation d'INformations Générales)..

Considéré sous l'angle des langages orientés objets, MERING peut être envisagé comme un langage issu de la logique SMALLTALK (distinction entre classes et représentants, contrôle par envois de messages, etc..). Il propose toutefois de dépasser ce formalisme pour introduire des orientations indispensables à la représentation des connaissances: notions de contraintes et de réflexes, possibilité, pour un type, de répondre aux mêmes messages que ses représentants, de disposer de valeurs propres, c'est à dire de donner aux types le statut d'entité au même titre que les représentants, sans devoir créer une notion nouvelle comme celle de méta-classe en SMALLTALK-80, etc..

Développé autour d'une idée d'uniformité (tout objet MERING est une **entité**, toute expression une transmission), Mering généralise la notion de **méthode** de SMALLTALK, en permettant le passage de messages, non seulement à l'entité toute entière, mais aussi à l'un quelconque de ses 'attributs'.

De plus, il incorpore un certain nombre de fonctions importantes pour la représentation des connaissances: héritage multiple, continuation, "réflexes", règles de production, etc...

MERING est écrit en VLISP mais, tout en conservant l'ensemble de l'environnement LISP (le langage MERING est transparent pour celui qui veut programmer en LISP), il s'impose

comme un langage autonome, indépendant du langage informatique qui a servi à l'implanter.

Dans le premier chapitre, nous expliquons les motivations qui ont été à l'origine de MERING, en décrivant la distinction entre formalisme objet et relationnel, et en proposant sept critères pour la construction d'un langage de représentation et de manipulation de connaissances.

A la lumière de ces critères nous examinons trois types d'approches utilisées en Intelligence Artificielle et qui souscrivent au formalisme objet: les réseaux sémantiques, les schémas (ou frames), et les langages orientés objet.

Dans le deuxième chapitre nous présentons le langage MERING et ses caractéristiques essentielles:

- 1 - Les sources de MERING
- 2 - Description de MERING
- 3 - les Transmissions
- 4 - Les entités

Une illustration des capacités du langage est décrite au chapitre III. Cette illustration comprend d'une part un certain nombre d'exemples d'application, et d'autre part des extensions au langage, la notion de classe et celle de règle de production:

- 1 - Illustration sommaire: application aux bases de données
- 2 - Questionnaire intelligent: un système de réservation de places de chemin de fer.
- 3 - Un jeu d'aventure
- 4 - Programmation par contrainte: application aux circuits électriques
- 5 - Type et Classe: deux notions distinctes
- 6 - La programmation par règles de production

Le chapitre IV offre une vision détaillée de l'implantation
actuelle du langage et comprend un manuel de référence de
MERING:

- 1 - Structure de l'interpréteur actuel
- 2 - Manuel de référence

CHAPITRE I

Le formalisme objet dans la représentation des connaissances

I.1 Généralités et historique

Autrefois considéré comme un aspect secondaire, la représentation des connaissances a donné lieu à des recherches de plus en plus importantes en Intelligence Artificielle, pour aboutir à la création d'un champ d'étude autonome, que les anglo-saxons appellent "cognitive modelling" ou "representation of knowledge".

A la fin des années cinquante et au début des années soixante, il s'agissait surtout d'améliorer les performances des programmes, en disposant de structures de données plus souples, plus efficaces.

Les systèmes à cette époque ne disposaient pas de beaucoup d'informations et l'essentiel des efforts portait sur la réalisation d'algorithmes généraux capable de résoudre n'importe quel problème: toute la difficulté revenait à trouver les heuristiques nécessaires pour explorer un domaine de recherche dans un temps admissible. De tels programmes se sont révélés inefficaces devant des problèmes faisant intervenir une grande

masse d'informations.

Les travaux en compréhension de langage naturel, et en "raisonnement de tous les jours" (common reasoning), renversèrent cette tendance. Schank, Winograd, Charniak et bien d'autres comprirent que les performances d'un programme provenaient tout autant de ses algorithmes que des informations qu'il détenait.

Cette prise de conscience entraîna une évolution de perspective. Cette nouvelle vision de l'Intelligence Artificielle mis l'accent sur trois types d'études: la représentation des connaissances qu'un système doit posséder pour être à même de remplir sa tâche, les divers moyens employés pour l'utiliser, et les mécanismes nécessaires pour faire interagir cette base d'informations avec le monde extérieur.

Cette transformation a été définie par PAPERT et GOLDSTEIN (Papert.76) comme le passage d'un point de vue caractérisé par une stratégie de la puissance (power based strategy) à une approche fondée sur la connaissance (knowledge based approach).

Le processus d'intelligence est alors déterminé par la connaissance du sujet, la structuration statique et dynamique de cette masse d'informations, et par les opérations qui s'appliquent à cette connaissance.

Il ne suffit donc pas de disposer de connaissances, encore faut-il les organiser: savoir retrouver une information pertinente, en acquérir une nouvelle, puis l'utiliser dans une tâche quelconque sont quelques unes des opérations fondamentales qui sont liées au problème de la représentation.

Quelques évènements importants jalonnent l'histoire de la représentation des connaissances.

En premier lieu, le mérite revient à QUILLIAN et à RAPHAEL, d'avoir introduit les premiers la notion de réseaux

sémantiques, considérés comme un ensemble de relations binaires superposées à un espace de noeuds, les concepts. Pendant plusieurs années, les réseaux sémantiques constituèrent la voie royale de la représentation des connaissances.

En 1972, le programme SHRDLU de WINOGRAD, causa un tel retentissement au sein de la communauté de l'Intelligence Artificielle, qu'il suscita bon nombre de recherches et de controverses: il fallait être pour ou contre SHRDLU; la polémique sur les représentations d'informations sous forme déclarative ou procédurale était engagée. Elle dure encore...

Un des nombreux intérêts de ce programme, est d'avoir appréhendé la connaissance sous forme de "théorèmes" procéduraux écrits en MICRO-PLANNER, dont la ligne de pensée directe cachait cependant un manque de modularité intrinsèque.

Quelques années plus tard, le premier système expert vraiment digne de ce nom, MYCIN, voyait le jour. Avec lui, toute une lignée de systèmes de représentation allait prendre son essor en consacrant l'emploi de connaissances déclaratives sous la forme de règles de production.

Peu de formalisme ont eu autant de succès que celui ci. Les règles de production sont employées un peu partout et deviennent de plus en plus le noyau central des débats actuels. L'industrie de pointe en a fait son cheval de bataille depuis la récente proposition des Japonais de créer des ordinateurs de la "cinquième génération".

Vers les années 1974-75, MINSKY et CHARNIAK décrivent leur conception concernant la manière dont les informations doivent être structurées, qui devait aboutir quelques années plus tard à une structure de représentation sous une forme très élégante: les schémas ("frames" en anglais).

Pour la première fois, les connaissances étaient regroupées sous forme de classes ou de types génériques, sortes de moules conceptuels servant à construire notre vision du monde. SCHANK, avec ses "scripts" aboutissaient à la même conclusion: nous concevons le monde à partir d'objets et de situations stéréotypées, qui nous permettent d'inférer des informations non

précisées dans la langue. Elles sont implicites pour les êtres humains parce que trop évidentes.

Aujourd'hui, la représentation des connaissances est devenue un domaine indépendant, qui requiert des efforts sur le plan informatique, mais aussi des réflexions interdisciplinaires où la philosophie, la psychologie et la linguistique ont leur mot à dire. En effet l'informatique ne résout pas les problèmes de logique et de contenu. Il faut donc passer par une phase préalable où les questions d'ordre purement conceptuelles doivent être résolues.

La représentation est une activité de conceptualisation qui s'articule autour de deux niveaux souvent confondus en pratique:

1 - Le niveau conceptuel qui s'attache à dégager les différentes catégories conceptuelles et les liens statiques et dynamiques qui s'élaborent entre les éléments de connaissances lors de l'accomplissement d'une tâche particulière.

2 - Le niveau logique qui correspond à la modélisation de ce que l'on appelle les "universaux" ou "connaissances a priori" en philosophie. Elles renferment toutes les notions fondamentales autour desquelles s'articulent les connaissances stricto sensu: la notion de classe, d'objet, de relation, de propriété, d'unicité et de pluralité, de nombre, de déduction, etc...

I.2 Le niveau conceptuel

Les chercheurs en Intelligence Artificielle ont formulé le problème de la représentation des connaissances à partir d'un point de vue pragmatique: un programme **comprend** une histoire s'il est capable de répondre à des questions posées à son sujet; il **sait** mener à bien une action s'il peut ordonner les opérations élémentaires nécessaires à son accomplissement. La connaissance est alors comprise comme un moyen, un intermédiaire entre un "input" et un "output".

De ce point de vue les représentations conceptuelles ne sont pas étudiées en tant que telles, mais dans une optique de performance et d'efficacité: quelle est la structure optimale de représentation des connaissances.

Dans ce cadre, les problèmes liés à l'organisation conceptuelle et ceux qui concernent leur utilisation deviennent étroitement liés. On dira qu'une structure est plus performante lorsqu'il s'agit de systèmes experts, ou qu'une autre devient préférable lorsqu'on analyse une histoire, etc... En effet, certains problèmes de représentation sont caractéristiques d'une classe d'application. Par exemple, les difficultés liées à la représentation temporelle ne concernent pas les systèmes experts, alors qu'ils constituent un aspect majeur de la compréhension des récits.

Cette approche donne lieu à des conceptions implicites, qui, bien que pouvant produire des résultats intéressants localement, donnent lieu à des schématisations et à des simplifications outrancières.

Cette constatation est particulièrement criante lorsque les représentations concernent les mentalités humaines. Les systèmes développés par SCHANK, par exemple, ont une vision très restreinte de l'esprit humain, et les modèles proposés tiennent plus de l'ordinateur que de l'homme. Mémoire à long et court terme, motivations réduites à l'état de buts, comportements simplifiés à l'établissement puis à la poursuite de plans, donnent l'impression d'évoluer dans un univers de robots, mécaniste à l'extrême, à l'image des animaux-machines de Descartes.

L'acte de définition, de construction d'un concept n'est pas une opération simple. Quelles sont les informations minimales à produire? Existe-t-il une approche qui soit plus féconde qu'une autre?

Cette réflexion a pour objet de distinguer les différents éléments susceptibles d'intervenir dans notre connaissance du monde, c'est à dire d'établir une véritable taxinomie, afin de dégager les catégories conceptuelles et surtout de mettre en évidence les différents liens (statiques et dynamiques) qui associent ces concepts entre eux.

Par exemple, la compréhension d'une petite histoire (un conte d'enfant) met en oeuvre un certain nombre d'informations qui pourraient être regroupées selon les types conceptuels suivants:

- les personnages
- les actions effectuées par ces personnages
- les buts, désirs, plans et volontés des personnages
- les objets de l'univers et les lieux
- les règles stipulant les lois de cet univers (ex: si l'on pousse quelqu'un il peut tomber)
- les situations standards (les scénarios) que l'on rencontre dans ce type d'histoire (ex: le "méchant" cherche à manger ou à faire périr le "bon")

etc...

S'agit-il de décrire précisément un aspect particulier d'un ensemble signifiant avec tout le degré de détail possible ou bien doit on chercher à donner une image d'ensemble, quitte à laisser de côté ce qui constitue souvent l'impression la plus importante quand une idée, une image, un phantasme se présente à l'esprit.

Le sens de 'journalisme' par exemple, ne renvoie-t-il pas à toute une nébuleuse de visions et de constructions mentales? Être journaliste c'est à la fois exercer une profession, mais aussi être reconnu comme tel, c'est à dire se comporter comme un journaliste, avoir les habitudes, les comportements, les manies, le langage d'un journaliste. Cela signifie l'écriture d'articles, la prise de position, mais aussi l'instabilité d'une profession...

Sur le plan de l'écriture, le journaliste se moule dans un style, une façon d'accrocher le lecteur, une manière très caractéristique d'assembler les mots et les phrases..

Un concept n'est jamais isolé. Il ne peut vivre seul. Il lui faut un espace où s'étendre, mais aussi d'autres concepts pour s'épanouir. Poursuivons notre exemple: 'journaliste' renvoie à des notions voisines: la presse, l'édition, les auteurs littéraires, les spectacles, la politique, le pouvoir de la presse,...

Tout cela fait partie du concept de 'journaliste', et nous vient à l'esprit lorsque nous évoquons des personnages de ce type ou la profession (en tant que modèle d'individu) dans son ensemble.

Evidemment, dans l'état actuel de nos connaissances, nous serions bien en peine de représenter toutes ces choses sur ordinateur, ce qui reste pourtant notre objectif.

Toute simplification doit être maîtrisée, tout raccourci contrôlé. Nous ne pouvons nous satisfaire de systèmes qui réduisent une chambre à "un endroit où l'on dort, où l'on a des rapports sexuels et où l'on change de vêtement" (Lehnert.83). Nous ne sommes plus à l'époque des pionniers de l'Intelligence Artificielle, lorsque toutes les informations, étaient rassemblées sur la p-liste des atomes, lorsque les "mondes de bloc" constituaient tout son univers.

De ce fait, il nous paraît fondamental de séparer les problèmes de l'analyse conceptuelle en soi de ceux de leur implantation sur ordinateur, les points théoriques concernant

l'organisation des concepts de leur performance pour une tâche particulière, la "compétence" conceptuelle de sa "performance" pour reprendre la terminologie de Chomsky.

L'analyse conceptuelle revient alors à la construction de modèles d'univers, qui rendent compte de l'organisation des concepts du domaine quotidien ou scientifique, c'est à dire de notre image du monde concret et abstrait en distinguant les problèmes d'organisation de ceux de traitement.

L'analyse conceptuelle doit être envisagée comme un champ autonome, associé cependant à la sémantique linguistique d'une part et à la psychologie cognitive d'autre part. Il semble que cette autonomie soit seule garante d'une vue scientifique de la conceptualisation et de son développement. De plus, il s'agit de dégager ces recherches du bricolage informatique, tout en considérant l'ordinateur comme outil privilégié par sa capacité à formuler des modèles conceptuels testables et donc de motiver des études dans ce domaine.

Ce que nous entendons par conceptualisation ne doit pas être pris au sens restrictif de l'étude statique des structures conceptuelles, mais fait intervenir les notions dynamiques que l'on résume souvent par le terme "raisonnement".

En dégagant les principes de la conceptualisation, il est possible d'aller plus loin. L'analyse conceptuelle conduit à l'établissement d'une véritable ontologie, à la caractérisations des lois qui régissent les objets et les phénomènes par l'intermédiaire de constructions et opérations conceptuelles, en définissant ce que R.THOM (après HERACLITE...) nomme le "logos" de la forme (Thom.80). Il est probable que nous ne pourrions bien faire "raisonner" les ordinateurs que lorsque nous serons nous même capables de dégager les différentes structures conceptuelles que nous manipulons quotidiennement et la plupart du temps sans en être véritablement conscient.

I.3 Formalisme objet et relationnel

Le niveau logique s'occupe des concepts de base sur lesquels toute représentation ultérieure sera construite. Pour les informaticiens, ce niveau correspond directement à la problématique qui régit leur démarche lors de l'implantation de langages de représentation et de manipulation de connaissances. C'est à ce niveau là que nous nous intéresserons dans la suite de cette thèse.

Il existe deux types de formalisme dans la représentation des connaissances. L'un met l'accent sur les relations qui existent entre les concepts, et l'autre sur les propriétés même de ces concepts. Nous intitulerons ces deux approches **formalisme relationnel** et **formalisme objet**.

Depuis ARISTOTE les choses de l'univers (puis les idées avec Platon), sont considérées comme des "objets" bénéficiant d'un certain nombre d'attributs: l'essence, la qualité, la relation, le temps, le lieu, la situation, l'action, la passion et l'avoir.

L'un de ces attributs occupe une place prépondérante dans les problèmes de conceptualisation, celui de relation. Dire que Paul est plus grand que Pierre ou que Jules est le père de Maire, c'est exprimer des relations, des rapports, entre des individus précis. Mais à cette époque, la notion de relation n'était pas encore très bien comprise. Elle était encore confondue avec celle d'attribut, il n'y avait pas de variables, pas de calcul sur les relations.

Autant de propriétés qui ne purent s'exprimer que plus tard vers la fin du 19ème et le début du 20ème siècle, avec les développements de la logique formelle, et les travaux de FREGE (Frege.1891), WITTGENSTEIN (Wittgenstein.1921), WHITEHEAD et RUSSEL (Whitehead.1910), et bien d'autres.

Bien avant les logiciens, au moyen âge, certains s'étaient posé le problème des "universaux", ce que nous appellerions aujourd'hui les "types" ou "classes". Il s'agissait de déterminer la place des notions générales par rapport aux objets particuliers: quelle est l'existence du concept 'chien'? A-t-il la même réalité que ces deux bons toutous que sont Médor et Fido?

Trois écoles s'affrontaient alors: les réalistes, les nominalistes et les conceptualistes. Pour les premiers, les choses sont réellement divisées en classes. Les universaux sont des objets tangibles au même titre que les choses qu'ils désignent. La nature n'est qu'un livre qu'il suffit de déchiffrer. Une opinion qui connut de beaux jours pendant le 19ème siècle.

Les seconds eurent une idée qui fut reprise par les positivistes logiques. Les universaux n'existent pas réellement. Ils ne sont qu'une commodité introduite par le langage. Cette vision exprime une défiance à l'égard des entités abstraites, et de toute terminologie désignant des êtres inobservables (Carnap.50). Elle affirme que l'univers se compose d'individus et de leur somme et de rien d'autre. Tout le reste n'est qu'affaire de langage, une manipulation d'énoncés, qu'il faut formaliser si l'on veut être sûr de dire des choses qui ont un sens.

Les systèmes de représentation élaborés selon ce principe répondent à ce que nous appelons le **formalisme relationnel**.

Enfin les derniers étaient partisans d'une voie médiane, plus psychologique. Les universaux bien que n'étant pas à proprement parler "réels", possèdent pourtant une certaine existence, plus évidente qu'une simple nominalisation: il s'agit des concepts, objets mentaux qui ne sont pas exactement des choses réelles, mais ne se réduisent pas non plus à des jeux d'affirmations.

Cette idée est en fait très moderne. Elle doit être rattachée aux travaux consacrés à la psychologie cognitive, dont Piaget fut l'un des principaux chef de file. Il a montré que l'individu effectue une véritable construction mentale du

, réel (Piaget, 45). La décomposition et le regroupement des objets en concepts par exemple, relèvent d'opérations de comparaison, de sériation et d'emboîtement qui ne se développent chez le sujet qu'au fil des années, dans la prime enfance principalement.

Le réel est traduit intérieurement, chez l'individu, sous la forme d'un ensemble de "schèmes", véritables abstractions d'opérations concrètes qui sont effectivement réalisées lorsqu'il s'agit d'objets réels. Il y a manipulation et représentation mentale, l'exercice d'une véritable simulation.

Pour s'en convaincre, il suffit de consulter les expériences qui ont porté sur la comparaison de schémas d'objets. Le temps de réponse pour savoir si les deux images correspondent à des objets identiques ou inversés, est proportionnel à l'angle de rotation nécessaire pour superposer les objets. L'individu les manipule mentalement afin de les mettre en correspondance. La pensée est donc projetée dans la chose et confondue avec elle.

Pour les psychologues, la pensée revient à simuler le réel, avec tout le lot de déformations et d'idéalisations qui peuvent s'y apposer. Même les neurophysiologistes commencent partager ce point de vue: il y aurait véritablement dans le cerveau une organisation neuronale qui rendrait compte "d'objets mentaux" (Changeux, 83).

Les systèmes conçus autour de cette approche souscrivent à ce que nous intitulerions le **formalisme objet**

S'il faut choisir entre nominalisme et conceptualisme pour l'écriture de systèmes de représentation, la seconde approche semble plus appropriée. Elle exprime une plus grande adéquation entre les structures cognitives humaines et artificielles. Nous devrions donc toujours écrire des systèmes qui simulent un comportement plutôt que d'écrire des moteurs d'inférences qui ne font qu'engendrer de nouveaux énoncés et manipuler des expressions formelles.

A l'écart de la philosophie, les systèmes de représentation

se sont développés en utilisant au mieux les possibilités de l'informatique: exprimer la connaissance sous la forme de données ou de procédures a donné lieu à une opposition connue sous le nom de controverse déclarative/procédurale.

A l'origine de cette controverse, les réactions de certains chercheurs à l'égard de travaux comme celui de WINOGRAD (Winograd.72). Dans son système SHRDLU, il maintenait la connaissance de l'univers du robot à l'intérieur de petits paquets procéduraux, des théorèmes, écrits en PLANNER (Hewitt.71). A l'époque, les critiques des tenants de l'approche déclarative n'étaient pas sans consistance. Les problèmes de modularité et de maintenance d'un gros corpus d'informations procédurales se heurtaient à des impératifs techniques.

La démarche procédurale, lorsqu'elle est employée sauvagement, est a priori lourde, efficace mais peu flexible, peu apte à incorporer aisément de nouvelles informations, de nouveaux schèmes. De là l'idée de mettre tout ensemble et en vrac l'ensemble des connaissances nécessaires à la réalisation d'un diagnostic, à la résolution d'un problème, ou toute autre tâche du même ordre. Il s'agit alors d'adopter un point de vue déclaratif et d'écrire un interpréteur général, un moteur d'inférence, dont la fonction est de gérer, pendant le raisonnement, l'ensemble des énoncés qui ont été placés dans cette base.

Si les motifs de modularité et de flexibilité sont très légitimes, il faut diminuer l'importance de la non-modularité et de l'interdépendance entre informations dans un système procédural. En MERING, comme dans tout langage d'acteur, tout est procédural. Même les règles de production, au niveau profond, sont représentées par des expressions qui accomplissent une action, et ne sont pas seulement interprêtées par un moteur d'inférence.

De plus de nombreuses spécifications peuvent être décrites, pour l'utilisateur, sous forme de connaissances déclaratives. Le système se charge alors de la compilation sous une forme profonde uniquement procédurale.

En effet, il ne faut pas confondre modularité fonctionnelle et modularité procédurale. La première revient à affirmer que

tout doit se passer comme si chaque élément du système informatique, en fait chaque connaissance, était complètement indépendant du voisin. Il s'agit d'un critère de modifiabilité: pour l'utilisateur, l'adjonction ou la suppression d'informations n'entraîne pas une réorganisation de la base. Le second va plus loin. Il stipule que l'aspect déclaratif et l'aspect procédural d'un système doivent être parfaitement séparés, que les connaissances ne doivent pas interférer avec l'interpréteur.

Cette conception est néfaste, car elle entretient une controverse là où elle n'existe plus vraiment. Il devient en effet de plus en plus difficile de savoir ce qui est procédural de ce qui ne l'est pas. Le "pattern matching" appliqué aux frames satisfait-il un mécanisme déclaratif, à l'instar des autres opérations de filtrage telle que l'unification par exemple, alors qu'il fait intervenir lors de son fonctionnement des déclenchements de procédures? Peut on dire qu'une contrainte, par exemple celle qui unit les trois termes d'une addition $Z=X+Y$, est déclarative si son expression prend la forme de "démons" attachés à des "slots" particuliers. Même les systèmes experts, les temples du déclaratif, utilisent de plus en plus des procédures activées lors de l'unification d'une règle.

Tout le monde est d'accord sur un point: les systèmes de représentation et de manipulation de connaissances doivent au moins agréer à la modularité fonctionnelle. Ensuite, savoir si l'on utilise un moteur d'inférence ou si l'on compile les règles sous la forme d'entités procédurales, est un choix technique où les problèmes déclaratif/procédural, avec leurs caractères polémiques actuels, ne doivent pas intervenir.

En revanche l'opposition nominaliste/conceptualiste demeure. Elle n'est pas fondée sur des critères techniques mais sur des conceptions concernant nos rapports avec le réel. Dans ce cadre, nous avons vu que la logique moderne est fondée sur le nominalisme. Les concepts n'existent qu'en tant que variables qui interviennent dans des énoncés logiques.

Par exemple dire que tous les humains sont mortels n'est pas l'expression d'une propriété caractéristique de l'entité 'être-humain' dont tous les individus sont issus, mais l'affirmation d'une inférence qui peut être réalisée pour tous

les individus qui portent l'étiquette 'humain', c'est à dire qui sont dans un certain type de relation (intitulé généralement 'isa'ou 'sorte-de') avec le terme 'humain'. Cette vision s'écrit en logique:

$$(x) \text{ isa}(x, \text{humain}) \Rightarrow \text{mortel}(x)$$

alors qu'elle se représente ainsi, suivant la thèse conceptualiste qui admet l'existence du concept 'humain'.

```
type humain
| mortel <- vrai
```

Ces deux approches sont radicalement différentes. La première est tournée vers une vision du monde sous la forme d'énoncés prédicatifs, alors que la seconde conçoit des entités qui disposent d'attributs. Par exemple exprimer un ensemble de propriétés sur une personne s'exprime suivant le formalisme relationnel comme une suite de prédicats, alors qu'il revient à la définition d'attributs attachés à un type dans le formalisme objet.

Cette opposition, qui, nous l'avons souligné, prend ses sources dans la discussion entre philosophie Aristotélicienne et positivisme logique, ne doit pas être confondue avec la controverse déclaratif/procédural. En effet, il peut être tentant d'associer l'aspect déclaratif au formalisme relationnel, et la vision procédurale au formalisme objet, ne serait ce que par l'attachement qui unit prédicat et logique d'une part, et objet et attribut d'autre part.

Mais les différents systèmes informatiques se chargent de réfuter cette affirmation. En effet PLANNER qui est un langage procédural par excellence, appartient à la vision relationnelle, alors que EMYCIN (Melle.81), système expert complètement déclaratif voit ses faits organisés sous la forme d'arbres contextuels qui ne sont en réalité qu'une structuration des connaissances sous le formalisme objet.

Ce qui caractérise un système organisé sous le formalisme relationnel, c'est le fait que les entités n'ont d'existence que par l'ensemble des propositions dans lesquelles elles interviennent. Cette position entraîne l'écriture de programmes qui sont fondés sur une gestion de ces relations, sous la forme de manipulations d'énoncés.

En revanche les systèmes construits autour du formalisme objet ne manipulent pas d'énoncés. Ils simulent un phénomène et réalisent une correspondance entre un monde extérieur et un monde intérieur dont il constitue un modèle en réduction. La compréhension d'un texte revient alors, non pas à ajouter des faits et à affirmer des relations, mais à exécuter la scène telle qu'elle est modélisée.

Le langage que nous avons développé, MERING, et qui est décrit au chapitre suivant, souscrit au formalisme objet. Toute représentation devra donc faire l'objet d'une modélisation sous la forme d'entités, et non sous la forme de prédicats. Par exemple, dans un système prédictif, les informations caractérisant deux individus se mettent sous la forme:

```
(sorte-de etre-humain pierre)
(sorte-de etre-humain marie)
(profession pierre ingénieur)
(profession marie avocat)
(age pierre 32)
(passions pierre (voile ski lecture))
(sexe pierre masculin)
(marié marie paul)
(sexe marie féminin)
(donne pierre marie livrel)
```

alors que les mêmes notions s'écrivent ainsi sous un formalisme objet (en prenant la syntaxe MERING):

```
(personne => pierre
  sexe    <- masculin,
  age     <- 32,
  profession <- ingénieur,
  passions <- (voile ski lecture),
)

(personne => marie
  sexe    <- féminin,
  profession <- avocat,
  marié   <- paul,
)

(don => donl
  agent <- pierre,
  destin <- marie,
```

thème $\leftarrow$ livrel,
)

On voit qu'il ne s'agit plus d'énoncés indépendants: l'ensemble des propriétés caractéristiques d'un individu sont placées au sein de la même entité. De plus, tout ce qui est considéré comme évènement, phénomène ou situation donne lieu à la création d'entités au même titre que les individus, les animaux ou les objets.

Ces entités, qu'il s'agisse de personnes, d'actions ou de processus ont une existence réelle, un espace propre dans la base de connaissance, et sont caractérisées par leurs propriétés statiques et dynamiques, c'est à dire aussi bien par les relations qu'elles entretiennent avec les autres entités de la base, que par leurs comportements, c'est à dire l'ensemble des opérations qu'elles sont capables d'effectuer sur la base.

I.4 Sept critères pour caractériser un langage de représentation et de manipulation de connaissances

La constitution d'un (bon) langage de représentation et de manipulation de connaissances, qui souscrit au formalisme objet, doit pouvoir répondre à l'ensemble des critères suivants:

- A) Les éléments de base de la connaissance sont décrits sous la forme d'**entités** (au sens large du terme) caractérisées par leurs **propriétés** ou **attributs**.

Par exemple:

un objet est décrit par sa couleur, sa taille, sa forme, son poids, sa substance, son utilité, etc..

Une personne est caractérisée par son nom, son état civil, sa profession, son âge, son poids, ses occupations, ses goûts, ses désirs, son passé, ses intentions, etc...

Une action est décrite par l'agent, le thème, le lieu

et le temps, l'instrument, la source et la destination de cette action, etc...

Un scénario est représenté par ses acteurs, les lieux et les objets concernés, la condition pour qu'il ait lieu, l'ensemble des événements et l'état résultant du déroulement du scénario, etc...

- B) chaque attribut est caractérisé par un certain nombre de **valeurs explicites** et **implicites**, **contraintes**, etc...

par exemple: l'agent de l'action vendre doit être une personne ou une entreprise.

L'âge d'une personne est un nombre.

Par défaut la langue maternelle des Belges Wallons est le Français.

- C) A ces attributs peut être associée la manière explicite de trouver les informations manquantes, ou ce qui doit être effectué lors de l'ajout d'une donnée à cet attribut. Souvent, ces informations sont placées sous la forme de **démons**, c'est à dire de procédures qui sont activées lors d'un événement intervenant à l'un des attributs. Mais cette formulation n'est pas la seule: envois de messages, règles de productions, etc.. sont quelques unes des solutions qui permettent de se libérer du concept par trop procédural des démons.

- D) Il existe des **types génériques**, les concepts, dont les individus, les objets, les actions, .. du monde réel sont des concrétisations, lesquels sont intitulés les **représentants** (ou "instances" dans le jargon informatique) de ces types.

par exemple: Toto est représentant du type 'personne'
 aller-l est représentant du concept 'aller', c'est à dire qu'il s'agit d'un déplacement localisé dans le temps et dans l'espace et pour lequel

les agents et les circonstants sont connus.

- E) Il est possible de caractériser un type particulier en termes de types plus généraux, et donc de créer un ordre général ou partiel dans la spécialisation des concepts. En d'autres termes, il est possible de dire que des concepts sont plus généraux que d'autres. Les concepts plus spécifiques comportent plus de propriétés, et les contraintes sur leurs attributs sont plus restrictives que pour les concepts situés plus haut dans la hiérarchie.

Par exemple: 'voiture' peut être considéré comme un concept issu des types plus généraux véhicule et élément de standing. En tant que véhicule la voiture impose une restriction sur le poids, la taille, le mode de déplacement (sur route ou pistes), les caractéristiques externes telles que le nombre de roues, le nombre de places, etc.. par rapport au concept véhicule en général, et lui permet de se différencier des autres types de véhicules.

A ces impératifs concernant la représentation des connaissances sous le formalisme objet, nous voudrions ajouter quelques contraintes au sujet de leur manipulation:

- F) Le langage doit être aussi un langage de manipulation des connaissances, sans devoir faire appel à un formalisme extérieur au langage. En d'autres termes la méta-connaissance doit pouvoir être décrite sous la même forme que la connaissance. Il est alors possible de décrire toute une application sans recourir à un autre langage de manipulation. Ce critère répond à un souci d'homogénéité et de cohérence.

Le modèle d'acteur, par exemple, répond à ce critère: chaque entité du langage est considérée comme un objet informatique qui reçoit des messages, et répond à ces

messages en en envoyant d'autres. Certains acteurs seront alors plus consacrés à la représentation, d'autres dédiés essentiellement au contrôle.

- G) Des structures de contrôle élaborées (agenda, règles de production, mécanismes de filtrage,...) doivent pouvoir être décrites dans le langage lui-même, afin qu'elles puissent être modifiées par l'utilisateur pour l'écriture d'une application particulière. Ce critère satisfait un besoin de cohérence et de modifiabilité.

A l'aide de ces sept critères, nous allons examiner un certain nombre de techniques de représentation utilisées en Intelligence Artificielle, et qui souscrivent au formalisme objet:

- Réseaux sémantiques
- Schémas ou Frames
- Langages orientés objet

I.5 Les réseaux sémantiques

La représentation des connaissances à l'aide de réseaux sémantiques date de 1968, année où deux chercheurs QUILLIAN (Quillian.68) et RAPHAEL (Raphael.68) introduisirent cette notion. Il s'agit d'ailleurs de la plus ancienne forme de représentation de connaissances sous le formalisme objet utilisée en Intelligence Artificielle.

Il est difficile de définir ce qu'est un réseau sémantique, chaque chercheur ayant une vue différente sur les capacités de son modèle (En particulier BRACHMAN (Brachman.79) propose un ensemble de critères pour classer les réseaux sémantiques). Cependant un certain accord se réalise sur la définition suivante:

Un réseau sémantique est constitué d'un ensemble de **noeuds**, représentants des objets des événements ou plus généralement des concepts, et d'**arcs** qui définissent des relations entre les noeuds (figure I.1) . Formellement les réseaux sémantiques sont décrits par des graphes colorés, dans lesquels chaque couleur correspond à une relation entre deux objets.

Les réseaux sémantiques peuvent aussi être envisagés sous la forme d'entités décrites par une liste de couples <attribut,valeur>: les attributs sont les relations, et les valeurs des pointeurs vers d'autres entités, c'est à dire d'autres noeuds.

Dans leur vision traditionnelle, les réseaux sémantiques répondent aux critères A, D et E. En d'autres termes, ils ne permettent pas d'inclure des valeurs implicites ni des contraintes; la manière de retrouver des informations manquantes est définie par l'architecture du système lui-même (c'est "câblé") et ne laisse aucune liberté au programmeur dans ce

sens; enfin les réseaux sémantiques nécessitent un interpréteur externe pour retrouver et manipuler la base de connaissance: généralement un langage fondé sur la logique du premier ordre, très semblable aux systèmes de requêtes dans un thésaurus.

Certains des points ont été revus sur des systèmes plus récents, en particulier NETL, KL-ONE et ARCHES.

NETL (Fahlman.79) est un système développé par FAHLMAN, sur la base d'une conception matérielle: il estime que son modèle est réalisable en circuits intégrés, et que la recherche des informations peut être effectuée en parallèle. Pour cela il considère que chaque noeud est en fait un processeur et que les relations ne sont que des lignes sur lesquelles circulent les informations d'un noeud à l'autre.

Ce modèle, très riche en capacités, inclut notamment une distinction entre type, le prototype d'un élément, et classe, la somme de tous les éléments qui correspondent à un certain type (voir le chapitre III.5 pour une discussion de cette notion).

Cependant son modèle est par définition très rigide: en particulier il est difficile de programmer des types d'héritages qui n'ont pas été prévus par le constructeur du système.

Le système KL-ONE de BRACHMAN (Brachman.77) introduit des valeurs par défaut, des contraintes de type sur les relations et sur les valeurs, tout un ensemble d'informations sur la manière dont interagissent les données. Il permet même d'attacher des procédures évaluables par l'interpréteur. Il est d'ailleurs à noter que de nombreux chercheurs estiment que KL-ONE tient plus des systèmes de Schémas que des réseaux sémantiques stricto sensu.

ARCHES de CHOURAQUI (Chouraqui.81) est un système de réseau sémantique qui propose des capacités de raisonnement déductif et analogique à l'intérieur d'un moule conceptuel très cohérent.

De plus, sa thèse présente une très bonne discussion des idées fondamentales qui sont à la source des réseaux sémantiques.

1.6 Les schémas

L'idée des schémas est issue d'un article de MINSKY (Minsky.75) et des travaux de CHARNIAK (Charniak.76) dans les années 1975-76. Ils estimèrent que la connaissance chez un être humain n'est pas disposée en vrac, mais se trouve structurée autour d'unités d'informations.

En particulier les événements de la vie quotidienne (prendre les transports en commun, conduire une voiture pour aller à son travail, acheter un billet d'avion, prendre un repas dans un restaurant, etc..) peuvent se représenter aisément à l'aide de schémas, c'est à dire d'entités de forme pré-définie, dont il suffit de remplir les cases laissées libres.

Les schémas sont des entités qui incluent des informations déclaratives et procédurales, caractérisés par une liste d'attributs qui sont des généralisations des couples <attributs,valeurs> des réseaux sémantiques.

Une schéma est alors formellement une liste de couples <attribut,{facette}> où {...} représente une liste, et chaque facette correspond à un élément de description d'un attribut. Ces facettes contribuent à la spécification de données implicites, de contraintes, de valeurs par défaut, et peuvent même contenir des démons, c'est à dire des procédures déclenchées lors de la manipulation (en lecture ou en écriture) des valeurs des attributs.

Depuis 1977, de nombreux systèmes utilisent des schémas pour représenter leurs connaissances, et des langages ont été créés à cet effet: KRL (Bobrow 77.b), FRL (Goldstein.77), AIMDS (Sridharan.78) et UNITS (Stefik.79) étant les plus connus. Ils répondent tous aux cinq premiers critères (A, B, D, C, D et E), plus ou moins exactement:

FRL est fondé explicitement sur la notion de liste d'attributs comme nous venons de le définir. L'emploi des procédures est complètement généralisé, chaque attribut pouvant déclencher des procédures. Le mécanisme d'héritage est à la fois souple et puissant puisqu'il est intégré dans l'un des attributs (en particulier il est possible de définir des options sur le mécanisme d'héritage).

Cependant FRL n'est pas un langage à proprement parler: il s'agit d'un ensemble de procédures écrites en LISP qui permettent de manipuler les schémas. Il faut donc considérer les schémas en FRL comme des structures de données puissantes, manipulables par un certain nombre de fonctions qui constituent les voies d'accès privilégiées à ces structures.

KRL est véritablement un langage de représentation par schémas très complet. Les schémas en KRL comportent des contraintes de types, des valeurs par défaut, et surtout des mécanismes d'invocations de procédures par **démons** et **servants**. Ce sont en fait des procédures LISP attachées aux attributs et invoquées lors de leur manipulation: les démons sont activés lorsqu'une valeur est introduite dans un attribut, alors que les servants indiquent comment trouver une valeur manquante.

L'héritage en KRL permet d'avoir une multiperspective, c'est à dire qu'un objet peut être considéré sous plusieurs aspects: par exemple, soit l'objet théâtre:

théâtre

perspective : établissement

perspective : représentation-artistique

Un théâtre héritera de caractéristiques différentes suivant qu'il est considéré sous le point de vue d'un établissement ou d'une représentation-artistique.

Il introduit des mécanismes de contrôle par agenda et par filtrage (pattern matching) très pratiques pour créer des applications complètes dans ce langage mais malheureusement figées (l'agenda est défini dans l'interpréteur du langage et non comme un schéma particulier).

AIMDS et UNITS sont dans la lignée directe de ces deux systèmes.

I.7 Les langages orientés objet

Les langages orientés objet proposent une vision nouvelle de la programmation en introduisant des concepts de structuration par **classes** et **représentants** ("instance" en anglais et dans le jargon des informaticiens) hérités du langage SIMULA, et un mécanisme de contrôle généralisé par envois de messages. Chaque objet, c'est à dire chaque élément du langage) est décrit par l'ensemble de ses champs locaux et par la liste de ses méthodes, c'est à dire des comportements qui sont évalués à la réception d'un message.

Les langages orientés objets, tels que SMALLTALK (Goldberg.83), PLASMA (Marcoux.79), ACT1 (Liebermann.81), OBJVLISP (cointe.82), GLISP (Novak.83), STROBE (Smith.83) et surtout LOOPS (Bobrow.82) répondent tous aux critères A, D, E, et G avec les restrictions suivantes:

La distinction entre classes et représentants est forte, et n'a rien à voir avec celle qui est proposée dans les langages de schémas.

Il est généralement impossible de rajouter des attributs (des champs locaux) ou des méthodes à un représentant particulier.

Les classes ne reconnaissent que les méthodes qui sont décrites dans la méta-classe 'class' (ou dans chacune des méta-classes associées aux classes comme en SMALLTALK-80). En particulier elles ne peuvent répondre directement aux mêmes messages que leur représentant.

Les mécanismes "d'instanciation" c'est à dire de création de représentants sont très figés (Briot.83): il s'agit toujours de créer un nouvel objet correspondant exactement au moule défini par la classe. En particulier il est impossible de créer une entité à partir d'une copie (exacte ou imparfaite) d'un représentant.

PLASMA et ACT1 ne souscrivent pas au critère E. Cependant ils introduisent des mécanismes de contrôle très puissants tels que l'appel par filtrage, la capture des continuations et la délégation de messages.

LOOPS (et STROBE qui en est une copie directe) est le plus développé des langages de ce type: il répond aux critères C et H, mais avec les restrictions suivantes: les procédures attachées aux variables (les attributs en LOOPS) ne peuvent être déclenchées que sur deux types de manipulations: si une valeur est affectée à la variable ou si elle est lue (cette procédure est déclenchée si une valeur manque ou après que la valeur ait été lue).

Il est possible de définir des valeurs par défaut, mais le langage ne comporte pas de contraintes.

Il est possible de programmer à l'aide de règles de production (Stefik.82), mais ces règles ne contiennent pas de variables.

La syntaxe entre objets et règles de production n'est pas uniforme: de ce fait ces règles de production semblent superposées au langage initial (cette sensation est due à la syntaxe, car en fait il y a réellement intégration entre les objets et les règles de production).

Dans le chapitre suivant nous présenterons de manière plus détaillée les principes mis en oeuvre dans le cadre des langages orientés objets.

I.8 Conclusion

Nous avons examiné les différents systèmes de représentation issus du formalisme objet. Nous avons vu que par rapport à nos critères concernant ce que doit être un bon langage de représentation, chacun des langages présentés ne répond qu'à un sous ensemble de ces critères.

C'est pourquoi nous avons développé un langage qui répond à l'ensemble de ces critères afin d'être à la fois un langage de représentation et de manipulation des connaissances.

CHAPITRE II

Le langage MERING

II.1 Les concepts de base

MERING est un langage d'acteur qui incorpore de nombreux mécanismes afin de le rendre particulièrement adapté à la représentation des connaissances dans des domaines tels que la compréhension du langage naturel et les systèmes experts.

Il est conçu principalement autour de deux notions:

- le contrôle par envoi de messages, et
- la représentation sous un formalisme 'objet'.

II.1.1 MERING: un langage d'acteurs

MERING est né de la synthèse de deux approches différentes dans le monde de l'Intelligence Artificielle. La première concerne les problèmes de représentation sous forme de 'schémas', dont les langages KRL et FRL sont les plus connus; la seconde provient d'une idée proposée par C.HEWITT (Hewitt.77) et A.KAY (Kay.76), qui introduisit un nouveau mécanisme de contrôle dans les langages de programmation. Celle ci donna lieu au développement des langages PLASMA et SMALLTALK. Ce dernier était en fait une sorte de SIMULA interprété, qui généralisait la notion de 'classe', et apportait une uniformisation de la syntaxe en considérant toute expression du langage comme une transmission.

Les systèmes informatiques supposent une certaine vision de ce qu'est (ou de ce que doit être) l'activité de programmation et l'organisation des programmes.

Les systèmes classiques traitent les programmes comme un ensemble de **données** sur lesquelles agissent des **procédures**. Les données sont la partie passive des logiciels alors que les procédures constituent leur aspect actif. L'exécution de ces programmes consiste à appeler, dans un ordre décrit par la structure des instructions, les procédures en leur fournissant les données nécessaires à l'accomplissement de leur tâche.

Ce type de programmation est encore prépondérant en informatique. Un effort important a été accompli dans les années 70 pour améliorer la structure des données, afin de simplifier la tâche des programmeurs, mais aussi d'être à même de repérer les erreurs dès la phase de compilation des programmes. Mais ces améliorations n'ont pas changé la vision d'ensemble: elles n'ont permis que de raffiner certains points critiques sans remettre en cause l'organisation des programmes.

Dans cette optique, les données et les procédures sont traitées indépendamment les unes des autres sans tenir compte des relations étroites qui les unissent. L'utilisation de réseaux sémantiques, par exemple, met en oeuvre une structure de données particulière (le réseau lui-même), et un

interpréteur pour le manipuler, créer de nouveaux liens, effectuer des requêtes, etc..

Il s'ensuit un style de programmation qui impose au programmeur des contraintes fortes (la distinction entre données et instructions) et le gêne dans son expression. Si une notion aussi intéressante que la programmation structurée permet de concevoir des logiciels bien construits et facile à "maintenir" dans des domaines où les nouveautés sont limitées (par exemple, l'informatique de gestion ou le calcul numérique), il n'en est pas de même en Intelligence Artificielle: la recherche requiert une liberté accrue et une potentialité plus forte des langages qu'elle utilise.

Pourquoi distinguer entre données d'une part et procédures (ou instructions) d'autre part? Pourquoi ne pas considérer que les programmes sont avant tout des ensembles d'objets informatiques, chaque classe étant caractérisée par les manipulations que ses objets connaissent, c'est à dire par leur sémantique. Ainsi, en contraste avec les systèmes classiques, les systèmes **orientés objets** sont fondés sur la reconnaissance d'une seule catégorie d'entité informatique, les **objets**, qui incorporent les aspect statiques et dynamiques au sein d'une même notion.

A partir du concept général d'objet, les structures de contrôle du langage peuvent s'engager selon deux voies différentes.

- La première revient à conserver la notion d'instruction et de procédures: les objets constituent alors une nouvelle catégorie d'éléments, mais l'architecture générale des programmes reste classique. C'est au programmeur de décider s'il préfère programmer "à la papa" ou bien recourir à ces nouvelles structures. Ce type d'organisation a été appliqué à des langages (le plus souvent compilés) comme SIMULA, qui incorpore la notion de **classe**, et ADA qui la connaît sous la dénomination de **package**.
- La seconde consiste à choisir une solution radicalement différente: considérer les objets comme les seules entités du langage et leur concevoir une structure de

contrôle n'ayant plus recours à la notion d'instruction, en introduisant la notion de **transmission**. les programmes ont alors pris le visage d'un ensemble d'**acteurs**, c'est à dire d'objets actifs, qui communiquent les uns avec les autres par **envois de messages**.

Aucun impératif ne vient plus limiter le langage: même la syntaxe est réduite à sa plus simple expression. Tout est ramené à une question de sémantique des acteurs et à la structure générale de contrôle. Le modèle d'acteur est alors défini comme la synthèse de la notion d'objet et de celle de transmission:

acteur = objet + transmission par envois de messages

Il constitue de ce fait une formalisation de ces idées et devient indépendant d'un quelconque langage de programmation.

Chaque acteur est issu d'un moule plus général qui constitue sa classe, et dont il est lui même l'un des représentants. Cette organisation, cette répartition des objets en classes d'une part et représentants d'autre part, est très pratique pour décrire des concepts: cette table ci est un représentant de la classe 'table', ce match de tennis est issu de la classe 'match-de-tennis'.

A cet égard SMALLTALK et PLASMA ont des conceptions différentes sur les classes. Le premier considère que les classes sont organisées hiérarchiquement, toute classe de niveau inférieur bénéficiant, par héritage, des caractéristiques du niveau supérieur, alors que le second traite cette question par l'envoi de messages à d'autres acteurs grâce à la notion de 'packager' (pomian.80).

L'un des tous premiers avantages des langages d'acteurs est d'avoir pratiquement éliminé la syntaxe (en la réduisant à une forme générale très simplifiée), et d'avoir transféré toutes les questions d'analyse sémantique dans les acteurs eux-mêmes. Des notions aussi simples que l'addition de deux nombres, la concaténation de chaînes de caractères, ou la synthèse de deux

éléments chimiques, peuvent s'écrire avec la même syntaxe sans devoir recourir à un analyseur lourd qui vérifie à chaque instant la nature des arguments en fonction de l'opération qui leur est appliquée.

Il suffit de savoir comment envoyer un message pour être à même de programmer. Par exemple, l'expression $4+5$, est une transmission. 4 est un objet de la classe des entiers, et $+5$ est un message qui lui est envoyé. Lorsqu'un entier reçoit un message désigné par l'en-tête $+$, il regarde l'élément qui suit. S'il s'agit d'un entier, il l'ajoute à lui-même et retourne le résultat: ici la valeur 9.

La concaténation de chaînes de caractères, "bon" + "jour" s'effectue selon le même principe. Le message $+$ (par abus de langage l'en-tête est souvent confondu avec le message lui-même) est alors interprété par la chaîne "bon" comme une opération de concaténation avec l'élément suivant, "jour". Pour ajouter un opérateur au langage, nul besoin de modifier l'interpréteur. Il suffit de créer une classe d'acteurs. Par exemple il serait possible de construire la classe 'élément-chimique' pour laquelle $Ag+NO_3$ signifie le mélange de deux constituants.

L'intérêt des langages d'acteurs est donc de simplifier la tâche du programmeur, mais aussi d'augmenter sa capacité à créer en définissant non pas une syntaxe et une sémantique stricte et contraignante, mais seulement un moule général sur l'organisation des objets et leur activation par envois de messages.

Les langages d'acteurs sont des outils privilégiés pour créer des systèmes de simulation. SIMULA, à l'origine, était un langage de simulation. Ses auteurs incorporèrent les notions de **classes** et **d'objets** pour faciliter l'écriture des programmes de simulation. Simuler c'est en effet créer des 'objets' (par exemple des bateaux, des guichets ou des personnes dans une file d'attente) issues de modèles généraux, les 'classes' (la classe 'bateau' est le modèle de tous les bateaux), et les manipuler selon un mode en rapport avec le comportement et la structure des choses réelles qu'ils représentent.

Il n'est donc pas étonnant d'avoir fait appel à la notion de langage d'acteur pour la réalisation d'un langage de représentation fondé sur le formalisme objet, lequel sous entend que toute compréhension résulte d'une simulation, et que toute conceptualisation revient, au niveau de la représentation, à "faire vivre" intérieurement un phénomène extérieur, comme nous l'avons signalé au chapitre I.

II.1.2 Un langage de programmation pour la représentation des connaissances

Autre source importante pour MERING, les langages de représentation des connaissances connus en Intelligence Artificielle. Ces derniers temps la mode est aux schémas et MERING n'a pu y échapper. Son père spirituel est bien sûr KRL de BOBROW et WINOGRAD. Il est dommage que très peu d'applications aient vu le jour en KRL. Mis à part GUS (Bobrow.77-a), le premier système développé par ces auteurs, aucune réalisation d'importance n'est sorti de ses entrailles.

En fait de nombreux chercheurs se sont mis à écrire des petits KRL, auxquels il manquait toujours quelque chose pour approcher l'original. FRL, développé au M.I.T par Goldstein et Robert est vraiment le parent pauvre des systèmes à schémas. Il ne s'agit ni plus ni moins que de "listes de propriété" (p-listes) généralisées, auxquelles ont été attachées des procédures. Et pourtant, malgré son dénuement par rapport à KRL (pas de notion de points de vue, pas de contrôle par agenda, et surtout pas d'unification par "pattern-matching"), ou peut être à cause de celui-ci, FRL a donné lieu à de nombreuses réalisations (Goldstein.77)(Rosenberg.80). Et pas seulement au MIT. Le système expert TOUBIB (Fargues.83), actuellement développé en France chez IBM, est organisé autour d'un langage de 'schémas' qui présente beaucoup de ressemblances avec FRL.

Cette notoriété de FRL est un peu dommage, car elle a nui, dans une certaine mesure aux schémas. Ce terme est devenu synonyme de structure de données disposant d'une faculté d'héritage, de valeurs par défaut et d'attachement procédural. Alors qu'il s'agissait pour MINSKY et CHARNIAK, les inventeurs

de cette notion, de tout autre chose: un point de vue sur la manière dont les connaissances doivent être incorporées dans un système "intelligent", ce que nous appelons le formalisme objet.

Prenons l'idée de valeurs par défaut, qui constitue la tarte à la crème de tous les nouvelles structures de représentation. Elle porte de nombreuses ambiguïté, principalement lorsqu'on l'associe avec la notion d'héritage. En effet, si les valeurs vraies sont héritées sans trop de problème il n'en est pas de même des valeurs par défaut qui induisent des difficultés liées au traitement de logiques non monotones (Winograd.80). Cependant il s'agit d'un mécanisme d'inférence si important qu'on ne peut plus s'en passer. En fait ce qui est en cause ce ne sont pas les valeurs par défaut, considérées comme une inférence d'un type particulier, mais la manière dont elle sont implantées sur la plupart des systèmes.

Dans la plupart des langages de représentation de type FRL, les parties dynamique des concepts, c'est à dire généralement les procédures attachées aux attributs (en anglais: slots), ainsi que la structure de contrôle générale, sont écrits en LISP, rendant le système de représentation dépendant d'un langage hôte qui lui est un véritable langage de programmation, avec lequel il est possible de réaliser une application complète.

Avec une telle démarche, on tend à mélanger les problèmes liés à la conceptualisation et ceux qui n'ont trait qu'au fonctionnement du langage LISP (Steels.79), et à retomber dans les errements de la programmation classique.

En outre, tout lien dynamique, toute structure de contrôle élaborée devant être réalisée dans le langage hôte (LISP généralement), chaque procédure devient rapidement une entité étrangère au sein de la base, et diminue d'autant les capacités d'autonomie, de généralité, de flexibilité et de modularité du système.

C'est pourquoi nous avons voulu que MERING ne soit pas seulement un langage de représentation augmenté de quelques

fonctions LISP, mais un véritable langage de programmation. Il se distingue en cela de KRL, FRL, KLONE, AIMDS,... pour qui la structure de contrôle générale est soit figée autour d'un mécanisme unique, tel que l'agenda en KRL, soit trop dépendantes du langage hôte. Ainsi, en MERING, les activations attachées aux attributs sont des transmissions écrites en MERING, et les structures de contrôle générales sont des entités MERING. Il est donc possible de construire ses propres mécanismes de supervision pour une application particulière, en considérant un agenda par exemple, comme une entité MERING, et non comme une structure de contrôle figée.

II.2 Description de MERING

II.2.1 Présentation générale

En Mering, à l'instar des autres langages d'acteurs, le programmeur développe une application en décrivant les messages à envoyer, et ce qui se passe lorsqu'un message est reçu.

Les messages sont décrits par des expressions, les transmissions, qui doivent se conformer à la syntaxe du langage. Une transmission élémentaire ne peut prendre que deux formes: la première est destinée à activer une méthode, il s'agit d'une M-transmission, et la seconde à déclencher un applicateur, c'est une A-transmission.

La syntaxe d'une M-transmission est la suivante (l'étoile indique que le nombre d'éléments est supérieur ou égal à zéro):

(<entite><selecteur><argument^{*}>)

et voici celle d'une A-transmission:

(<entite><attribut><selecteur><argument^{*}>)

Toute transmission est une succession d'évaluation de transmission élémentaires. Par exemple:

```

: (type => $objet personne
  tache      $dom tache
             $si-req (% # journaliste ? | tache ?),
  fonction   $dom profession,
  statut     $valeurs (employe agent-maitrise cadre)
             $mode $uni,
)
= PERSONNE

: (personne => pierre
  tache      <- '(rediger-article mener-enquete)
  fonction   <- journaliste
  statut     <- cadre
)
= PIERRE

```

La première transmission décrit le type **personne** comme sous-type de **\$objet** (un type système pré-défini). Cette définition est réalisée sous la forme d'une transmission où chaque attribut se voit décrit sous toutes ses facettes: domaine de définition, ensemble des **valeurs** permises, **mode** (indique si un attribut peut prendre plusieurs valeurs ou non), et réflexes de type **\$si-req** (si-requis) qui seront déclenchés à la réception du message de lecture.

Les symboles % et # sont des symboles réservés du langage et désignent respectivement 'cet objet' (comme SELF en SMALLTALK ou THIS en SIMULA) et 'cet attribut'. Le caractère \$ situé devant certains indicateurs indique qu'il s'agit de mots clés du langage.

La seconde transmission décrit Pierre comme une nouvelle personne dont les tâches sont de rédiger des articles et mener des enquêtes, dont la fonction est d'être journaliste, et qui possède un statut de cadre. Toutes ces informations sont décrites au cours d'une transmission complexe, qui s'apparente à la "cascade" de SMALLTALK.

La première transmission élémentaire crée l'entité **Pierre**, laquelle est retournée après création pour devenir le récepteur de la deuxième transmission. Celle ci envoie le message <- à l'attribut **tache** de Pierre, au cours d'une A-transmission. Le résultat de cette évaluation est encore **Pierre**, qui devient récepteur des transmissions suivantes.

Il existe plus d'une cinquantaine d'applicateurs élémentaires, mais les plus fondamentaux sont <=, <-, ?, del. Ils correspondent respectivement à l'affectation, l'ajout, la lecture, et la suppression de valeurs associées à un attribut.

Chaque attribut comporte une ou plusieurs facettes qui sont la contrepartie active des applicateurs. Ces facettes contiennent toutes les informations nécessaires à la description d'un attribut. Certaines d'entre elles contiennent les **réflexes**, comportements spécifiques liées aux attributs, et qui sont déclenchés lors de la réception d'une A-transmission (transmission applicateur).

Par exemple, à l'applicateur <- est associée la facette \$si-ajout, à l'applicateur ?, les facettes \$si-lu et \$si-req (activées suivant qu'il existe ou non une valeur dans l'attribut); enfin l'applicateur del active les transmissions attachées à la facette \$si-supr, si elle existe.

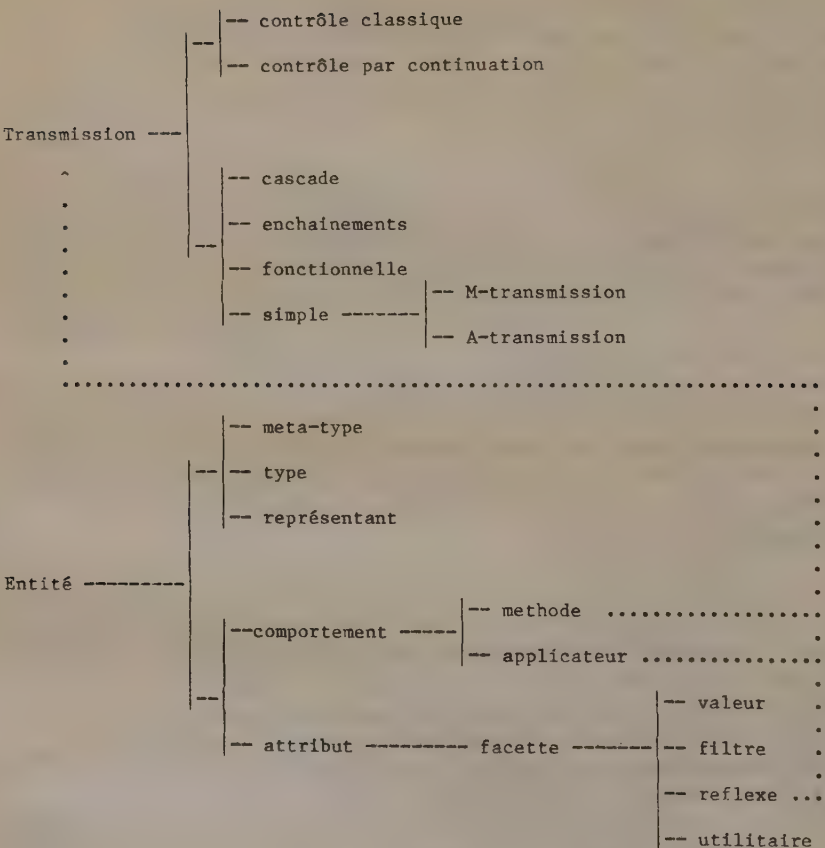
Il est possible non seulement de créer de nouveaux applicateurs, mais aussi de définir de nouvelles facettes que ces sélecteurs seront à même d'utiliser et d'activer.

Les applicateurs sont des comportements qui spécifient l'accès à un attribut. Dans la plupart des langages de programmation, l'accès à une variable est implicite. Le langage sait s'il s'agit de l'adresse ou du contenu de la variable. Par exemple l'affectation à la PASCAL:

A := B

signifie que l'on prend la valeur de B (accès en lecture) et qu'on la place dans l'endroit réservé à la variable A (accès en affectation). En MERING, l'accès aux attributs doit être précisé. Cette indication est nécessaire pour pouvoir déclencher des réflexes en rapport avec cet accès. Supposons qu'il existe deux entités nommées FOO qui possède respectivement un attribut A et un attribut B. L'affectation de la valeur de B à A s'exprimera:

Figure 2.1 L'organisation des éléments de MERING



(FOO1 A <- (FOO2 B ?))

où B est en lecture et A en affectation.

II.2.2 MERING en 13 points

La figure 2.1 montre la relation qui existe entre les différentes composantes du langage. Sa structure peut être décrite par l'ensemble des énoncés suivants:

- 1- Tout élément de l'univers MERING est une **entité**, toute expression une **transmission**.
- 2- Toutes ces informations sont placées sur des **facettes** prévues à cet effet. Il existe donc des **facettes** **filtres** qui caractérisent l'ensemble des contraintes auxquelles est soumis l'attribut, des **facettes** **valeurs** (valeur vraies et valeur par défaut), des **facettes** contenant les réflexes et des **facettes** **utilitaires**.
- 3- Les entités **communiquent** entre elles par l'envoi de messages.
- 4- Il existe deux sortes de transmissions élémentaires: la **M-transmission** envoie un message à une entité dans son ensemble; la **A-transmission** envoie un message à un couple <entité,attribut>.
- 5- Des transmissions plus complexes sont réalisées à partir de ces transmissions élémentaires: cascades, enchaînements et liaisons par fonctionnelle.

a) la **cascade** MERING correspond à une succession de transmissions dans laquelle l'entité résultat de la première transmission devient le receveur de la suivante.

b) L'**enchaînement** correspond à une suite de transmissions détachées les unes des autres. (semblable au PROG N LISP, et à la suite des instructions dans les langages de programmation).

c) Les **fonctionnelles** sont des entités qui règlent le contrôle, l'évaluation et la manipulation des environnements dynamiques en MERING.

6- Deux niveaux de transmission sont disponibles: le premier, traditionnel, est géré directement par l'interpréteur. Le second revient à casser cet enchaînement préétabli, pour activer d'autres entités. ce mécanisme, intitulé **continuation**, permet de créer des structures de contrôle plus élaborées, tels que les dialogues entre "experts".

7- Toute entité est soit un **type** générique, soit un **représentant** d'un type. Les types eux-mêmes sont actualisation du méta-type **type**.

8- Il existe une hiérarchie entre les types génériques, conduisant à un héritage des comportements et attributs. Cet héritage peut être multiple: il s'agit donc d'un treillis et non d'une arborescence. A cet égard, il existe une priorité dans l'héritage. Une entité est donc toujours plus d'un certain type que d'un autre. Cependant il est possible de spécifier le point de départ de la recherche de l'ascendant et donc de court circuiter ce mécanisme de priorité.

9- Décrire un type signifie:

a) énumérer l'ensemble des **comportements** de ses représentants, et du type lui-même, activables par l'envoi de messages;

b) définir l'ensemble des entités locales qui lui sont

associées: ses **attributs**.

- 10- Les **comportements** d'une entité sont de deux types: les **méthodes**, associées à des messages envoyés à l'entité toute entière, et les **applicateurs**, actions entreprises lors de l'envoi d'un message à un attribut, ce dernier en devenant le récepteur. Les mêmes messages peuvent être envoyés au type ou à l'un de ses représentants
- 11- Les **attributs** sont caractérisés par leur domaine de définition, leurs valeurs par défaut, ainsi que par les comportements activables lors d'un déclenchement de cet attribut par l'envoi d'un message spécifique: lecture, écriture, suppression, etc...
- 12- L'environnement de programmation de MERING est constitué d'un ensemble de types et de représentants pré-définis: cellule, séquence, règles, filtres, base-de-travail, etc...
- 13- Programmer en MERING revient à décrire l'ensemble des types génériques, à créer quelques représentants de base, puis à réaliser une transmission initiale.

II.2.3 La syntaxe de MERING

Nous utilisons les notations suivantes:

/ indique un choix
 ex: $A = B / C$ signifie que A se réécrit en B ou en C
 {} indique zéro ou plusieurs occurrences d'un terme
 Les termes en **gras** signalent des mots réservés du langage

```

transmission  = ( expression )
               / litteral
expression    = fonctionnelle
               / expr-cascade
               / enchainement
               / expr-simple
               / fonction-LISP
fonctionnelle = litteral {argument}
expr-cascade  = expression | suite-cascade
suite-cascade = expression
               / expression | suite-cascade
enchainement  = expression & suite-ench
suite-ench    = expression
               / expression & suite-ench
expr-simple   = A-transmission / M-transmission
A-transmission = receveur attribut selecteur {argument}
M-transmission = receveur methode {argument}
receveur      = nom-entite
attribut      = selecteur
selecteur     = litteral / sel-super / transmission
sel-super     = (super: litteral)
               / (super: litteral nom-entite)
argument      = litteral / transmission
litteral      = nom-entite / chaine / nombre / atome
nom-entite    = atome
    
```

De plus, un certain nombre d'abréviations sont proposées:

```
(ce obj)          = (% obj ?)
(ce obj liste)    = (% obj ? | liste)

^obj              = (cell => obj)

@obj              = (obj ?)
```

Ex:

```
: '(ce profession statut)
= (% profession | statut)

: (^a <- truc & print)
  (CELL => A
    CELL <- (TRUC),
  )
= A

: @a
= TRUC
```

II.3 Les transmissions

La syntaxe de MERING est gouvernée par la notion de transmission. Il s'agit du mécanisme de base de contrôle implanté au sein du langage, toute opération étant réalisée de cette manière. En particulier la définition des types et la création des représentants s'effectuent de cette manière.

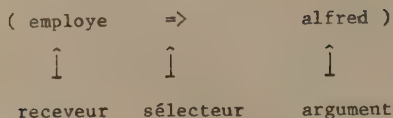
Cependant, dans sa version actuelle, MERING a été écrit en GLISP (Greussay.78), et toute fonction LISP est disponible à n'importe quel moment sans aucune modification, permettant ainsi une ouverture "vers la bas" sur le langage d'implantation. En effet une transmission en MERING ressemble beaucoup, quant à son apparence, à une forme LISP. Toute transmission commence et finit par des parenthèses. Mais l'analogie de syntaxe s'arrête là, car l'ordre des éléments est totalement différent.

I.3.1 A-transmissions et M-transmissions

Une transmission est caractérisée par l'envoi d'un message à une entité ou à un attribut qui sert de receveur. Les deux transmissions élémentaires sont la **M-transmission** (transmission méthode) et la **A-transmission** (transmission opérateur).

La première correspond à l'envoi d'un message qui a pour but l'activation d'une **méthode**, c'est à dire d'une expression, connue directement, ou indirectement par héritage, à cette entité. Syntaxiquement une telle transmission s'écrit désignant l'entité receveuse, le nom du message (appelé aussi lecteur de la méthode), et enfin ses arguments. C'est ce type de transmission que l'on retrouve dans la plupart des langages actuels, en particulier SMALLTALK, GLISP, LOOPS, et dans une certaine mesure PLASMA et ACT1. La création d'un représentant

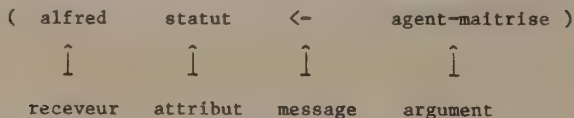
suit ce schéma. Par exemple:



est une M-transmission dont le sélecteur est '=>' et signifie que le receveur doit engendrer un nouveau représentant de son type.

La A-transmission est plus spécifique au langage MERING et constitue une originalité par rapport aux langages d'acteurs traditionnels. Nous étudierons plus attentivement ses propriétés lorsque nous étudierons les attributs, leurs facettes et leurs réflexes.

Syntaxiquement, la A-transmission est caractérisée par une entité et un attribut receveur suivit du nom du message et de ses arguments. L'affectation et la lecture de la valeur d'un attribut sont deux des A-transmission les plus usitées:



est une A-transmission qui affecte la valeur **agent-maitrise** à l'attribut **statut** de l'entité **alfred**.

Une A-transmission consiste à envoyer un message, non pas à une entité toute entière, mais seulement à l'un de ses attributs. Ici le receveur n'est plus une entité, mais un couple (entité attribut).

L'envoi d'un tel message déclenche une transmission locale à l'entité et connue sous le nom d'applicateur. Mais l'action de cette transmission dépend aussi de l'activation des différents réflexes qui sont associés à l'attribut receveur, et qui modifie

donc le résultat de l'applicateur. Par exemple:

```
: ($objet => foo
  A      $dom $nombre
        <- 2
        $si-lu (% B ?),
  B      $dom $nombre
        <- 4,
)
= FOO

: (foo A ?)
= 4
```

L'action de l'applicateur de lecture ? se voit modifiée par le réflexe `$si-lu` qui, en court-circuitant le mécanisme normal de lecture, ramène la valeur de B, et non celle de A comme l'indique l'expression. Les réflexes sont décrits plus amplement au chapitre II.4.3

Les applicateurs sont surtout utilisés pour créer de nouvelles structures "systèmes". Expliquons nous: le développement d'un logiciel en I.A. suit généralement deux phases. Au cours de la première sont conçues les procédures générales nécessaires à toute une classe d'applications (par exemple le moteur d'inférence d'un système expert), alors que la réalisation proprement dite, c'est à dire l'adaptation du système général à un cas particulier, s'effectue dans une seconde étape.

Les applicateurs sont alors élaborés au cours de la première phase. Ils permettent l'extension du langage à de nouveaux domaines. Ainsi la compilation de contraintes d'égalités sur les attributs sous la forme de réflexes, est réalisée par un applicateur au chapitre III.3, en prenant pour exemple des propagations de contraintes à l'intérieur d'un circuit électrique.

II.3.2 Cascades et Enchaînements

Les transmissions simples sont organisées sous la forme de transmissions plus complexes: **cascades**, **enchaînements** et **fonctionnelles**.

La succession des transmissions est gérée par un mécanisme de contrôle intitulé **cascade**. Il s'agit de permettre à une valeur retournée par une transmission précédente, de devenir le nouveau receveur d'un message. En effet l'évaluation d'une transmission ramène toujours un résultat, qu'il s'agisse d'une entité, d'atomes littéraux, de s-expressions ou tout simplement des deux symboles T et NIL. (le sens ici est légèrement différent de la cascade SMALLTALK. Dans ce langage, la cascade sert à envoyer un ensemble de messages à un même receveur. En fait, la cascade MERING peut souvent être employée au même effet, à cause des différences entre ces deux langages: en SMALLTALK, l'affectation retourne la valeur, alors qu'en MERING elle retourne l'entité receveuse).

Pour effectuer une cascade deux solutions sont possibles. Soit utiliser le parenthésage afin d'écrire des expressions "à la LISP", ou bien employer le symbole '|', qui sépare deux transmissions reliées par cascade. Par exemple:

```
(alfred role <- comptable | statut <- agent-maitrise)
```

est équivalent à

```
((alfred role <-comptable) statut <- agent-maitrise)
```

Dans ces deux cas, la valeur retournée par la première expression est l'entité 'alfred' qui devient le receveur du message suivant.

Il est possible de mettre à la suite l'une de l'autre deux transmissions sans qu'il y ait de rapport entre elles, à la manière d'un bloc PROGNG de LISP, et réaliser ce que nous

appelons des **enchainements**. Pour ce faire, il suffit de séparer les envois de messages par le caractère '&'. Par exemple les deux transmissions précédentes peuvent également s'écrire:

```
(alfred role <- comptable & alfred statut <- agent-maitrise)
```

II.3.3 Les fonctionnelles

Il ne suffit pas de pouvoir envoyer des messages. Encore faut-il être capable de le faire conditionnellement ou répétitivement. A cet égard des actions de contrôles ont été incorporées dans MERING. Certaines d'entre elles ne sont que des adaptations bien connues de fonctions ou macros LISP. D'autres en revanche sont plus spécifiques à MERING, et correspondent à la répétition d'opérations sur un espace, un domaine d'entités, bien défini.

En MERING, il existe quatre types de fonctionnelles:

- les fonctionnelles de contrôle, qui ont pour but de rompre le déroulement séquentiel des transmissions.
- Les fonctionnelles de quantification permettent d'appliquer une suite de transmissions à un ensemble d'entités.
- Les fonctionnelles d'évaluation, servent non seulement à évaluer des transmissions mais aussi à établir des **environnements locaux** dans le cadre desquels seront évaluer les transmissions.
- Les fonctionnelles d'entrées-sorties,

II.3.3.1 Les fonctionnelles de contrôle

Les fonctionnelles de contrôle sont très semblables dans leur syntaxe et leur sémantique à leurs homologues LISP.

- **IF** : il s'agit de la structure de sélection bien connue. La syntaxe est:

```
(if <condition> <S1> <S2> ... <SN> )
```

Si la valeur de l'évaluation de la condition est différente de NIL, IF ramène la valeur de l'expression <S1>, sinon les différentes transmission <S2>...<SN> sont évaluées en séquences. Cette fonctionnelle permet donc de construire des structures de contrôles de la forme SI <condition> ALORS <S1> SINON <S2>...<SN>.

Ex:

```
: (if (alfred role = comptable)
      (alfred revenu <- 7000)
      (alfred revenu <- 6000)
      & alfred revenu ?)
= 7000
```

- **WHILE** permet de réaliser des structures répétitives: boucles conditionnelles ou boucles infinies. Sa syntaxe est:

```
(while <condition> <S1> ... <SN>)
```

Tant que la valeur de l'évaluation de la condition est différente de NIL, les expressions <S1>...<SN> sont évaluées en séquences. Ex:

```
: (a <- '(a b c d))
= A

: (while (a ?) (a ! | print))
A
B
C
D
= NIL
```

- **AND** et **OR** ont un comportement identique aux **AND** et **OR** du langage LISP. Les différentes transmissions placées derrière ces deux fonctionnelles sont évaluées successivement. Dans le cas de **AND**, dès qu'une expressions ramène une valeur différente de **NIL**, l'évaluation est stoppée et la valeur **NIL** est retournée, sinon la valeur est **T**. **OR** en revanche ramène **T** dès que la valeur d'une des expressions est différente de **NIL**. Ex:

```
: (a <- toto & b <- riri)
= B

: (and (a = toto)(b = riri))
= T

: (or (a = riri)(b = toto))
= NIL
```

Ces fonctionnelles ne sont pas limitatives. Elles expriment seulement les structures minimum nécessaires à la programmation. D'autres structures de contrôles peuvent être aisément construites à partir de celles-ci ou directement en LISP.

II.3.3.2 Les fonctionnelles de quantification

Les fonctionnelles **FORALL**, **EXIST** et **TESTALL** sont utilisées pour réaliser des opérations répétitives sur un domaine précis. Elles expriment une analogie avec les quantificateurs logiques 'pour tous' et 'il existe'.

FORALL applique successivement la transmission située dans la partie <expression> à tous les éléments du domaine de définition, et retourne la liste des résultats non nuls (à l'image d'un MAPCT en VLISP).

EXIST applique successivement la transmission de tous les éléments du domaine de définition, jusqu'à ce qu'un des résultats de l'évaluation retourne une valeur différente de NIL. La fonctionnelle retourne ce résultat.

TESTALL à un fonctionnement semblable à **FORALL**. Si tous les résultats des évaluations successives sont différents de NIL, **TESTALL** ramène T sinon il retourne NIL.

Leur syntaxe est la suivante:

(nom-fonctionnelle variable domaine expression)

La variable n'est en fait qu'un argument formel qui sera remplacé par une entité (ou un objet quelconque, atome ou s-expression) appartenant au domaine de variation. Celui ci peut être un ensemble d'entité, tous les représentants d'un type, ou simplement les valeurs d'un attribut.

Ex:

; La méthode 'allinst' ramène la liste de tous les représentants
issus d'un type et de tous ses sous-types ;

```
: (forall x (employe allinst)
  (if (x statut = cadre)
    (x agence <- APEC)))
= ALFRED
```

```
: (a <- '(1 2 3 4 5 6 7))
= A
```

```
: (exist y (a ?)
  (if (y > 3) (y print)))
4
= 4
```

a fonctionnelle EXIST s'arrête dès qu'elle rencontre une
valeur supérieure à 3. Si l'on écrit FORALL à la place de EXIST,
les valeurs 4 5 6 7 sont affichées:

```
: (forall y (a ?)
  (if (y > 3) (y print)))
4
5
6
7
= (4 5 6 7)
```

```
: (testall z (a ?)
  (z num?))
= T
```

```
: (testall z ('(2 4 Y U 6 8) ?)
  (z num?))
= NIL
```

Ces fonctionnelles sont très employées pour toutes les
opérations de recherches sur un ensemble d'entités, dans les
règles de production notamment. Leur intérêt vient aussi du
fait que l'élément du domaine de variation est substitué à la
variable, ce qui permet de placer celle ci n'importe où dans une
transmission, à la place du receveur bien sûr, mais aussi à
celle du sélecteur ou des arguments.

D'autre part ces fonctionnelles peuvent s'emboîter pour

former des boucles imbriquées. Il faut alors s'assurer que les variables portent des noms différents pour ne pas créer de conflit. Ex:

```
: (a <- '(2 4 6 8) & b <- '(1 2 3))
= B

: (forall x (a ?)
  (forall y (b ?)
    (x + y | prinl
      & terpri )
    & T )
345
567
789
91011
= T
```

II.3.3.3 Les fonctionnelles d'évaluation

Ces fonctionnelles servent à l'évaluation d'une transmission, mais aussi à établir des environnements locaux.

Fonctionnelles d'évaluation

La fonction de base de MERING est MEVAL, qui évalue une transmission et retourne son résultat. Pour l'évaluateur, et dans l'implantation actuelle, toute forme est soit une transmission MERING soit une expression LISP. De fait, sous MERING, l'utilisateur a directement accès à tout l'environnement LISP.

Ex:

```
; addition style MERING ;  
: (meval '(2 + 3))  
= 5  
  
; (addition style LISP ;  
: (meval '(+ 4 5))  
= 9
```

D'autres fonctionnelles d'évaluation ont été définies, et sont directement semblables à leurs quasi homonymes LISP:

- (MPROGN <S1>..<>Sn>) évalue en séquence les différentes expressions <S1>..<>SN> et retourne la valeur de la dernière évaluation.
- (MEVPROGN <L>) évalue séquentiellement tous les éléments de la liste <L> et retourne la valeur de la dernière évaluation. etc...

Fonctionnelle de création d'environnements locaux

La fonctionnelle de manipulation d'environnement **MLET** change temporairement l'environnement pour définir des entités locales. Sa syntaxe est la suivante:

(**MLET** <L> <S1>..<>Sn>) où <S1>..<>SN> est une liste de transmissions, et <L> une liste de la forme:

- () dans le cas où il n'y a pas de définitions locales. Dans ce cas la sémantique de **MLET** devient identique à celle de **MPROGN**
- (<ENT1> <TYPE1> ... <ENT2> <TYPEn>) autrement.

La fonctionnelle **MLET** crée dynamiquement et en parallèle un ensemble de représentants <ENT1>..<>ENTn> de types respectifs <TYPE1>..<>TYPEn>. Si des entités existait déjà sous ce nom, elles seront masquées par les nouvelles entités ainsi créées. A la sortie de cette fonctionnelle, l'environnement initial est restauré.

Le type des entités est quelconque, en particulier ces entités peuvent être de type **var** pour créer des pointeurs locaux vers d'autres entités. Lorsque le type n'est pas défini,

ces entités sont supposées être issues du type \$objet.

Ex:

```
: (seq => a | <- '(q w e r t y))
= A
```

```
: (a ?)
= (Q W E R T Y)
```

```
: (mlet ((a )(b seq)(c var)) ; a est de type $OBJET ;
      (a truc <- 'chose      ; b est de type SEQ      ;
      (b <- '(a z e r t y))
      (c := a)                ; c est une variable qui
contient a;
      (c print)
      (b print))))))
($OBJET => A
  TRUC <- (CHOSE),
)
```

```
(SEQ => B
  CELL <- (A Z E R T Y),
)
= B
```

; l'environnement initial est restauré ;

```
: (a print)
(SEQ => A
  CELL <- (Q W E R T Y),
)
= A
```

```
: (b print)
= NIL
```

```
: (c print)
= NIL
```

La fonctionnelle MLET nous semble très importante, et pourra conduire à des développements ultérieurs, tels que la construction de types locaux, et ainsi parvenir à une vision plus dynamique de la notion d'entité.

II.3.3.4 Les fonctionnelles d'entrées-sorties

Dans l'état actuel de l'implantation, les fonctionnelles d'entrées-sorties ne sont purement et simplement que les fonctions LISP classiques: PRINT, PRIN1, READ et TERPRI.

II.3.4 Les continuations

Outre le mode normal d'évaluation, MERING propose un mode contrôle dans lequel une transmission spécifie sa continuation. Cette notion de continuation provient en droite ligne de la sémantique des lambda-langages (Greussay,77) et du langage PLASMA (Durieux,81).

Une fonctionnelle **send:** a été créée à cet effet. Elle a pour objet d'exécuter la transmission qui lui est passée en argument, après écrasement de la pile. De cette manière, il y a retour au niveau de la boucle supérieure (au "toplevel"), et évaluation de la transmission dans l'environnement global.

Ex:

```
(a <- '(a d f g) | print)
(SEQ => A
  CELL <- (A D F G)
)
= A

: (mlet ((a personne))
  (a profession <- medecin)
  (a print) ; A est imprim{ dans l'environnement cr{{ par MLET;
  (send: (a print)) ; A est imprim{ dans le contexte global ;
  (" salut" print)) ; Cette transmission ne sera pas évaluée ;
```

```
(PERSONNE => A
  PROFESSION <- MEDECIN,
)

(SEQ => A
  CELL <- (A D F G)
)
= A
```

Avec une telle technique, il est possible de reconstruire la boucle externe en s'inspirant de MICRO-PLASMA (Cointe.81) :

```
(seq => top
  ; la fonction de lecture continue sur l'{valuation ;
  read methode: () (% <- (read) & send: % eval),

  ; l'{valuateur reprend l'{valuation MERING et continue sur
  ;
  ; l'impression ;
  eval methode: () (% <- (meval (% ?)) & send: % mprint),

  ; et mprint retourne sur read: la boucle est bouclée ;
  mprint methode: () (% ? ù printat & send: % read),
)
= TOP
```

Il suffit ensuite de taper le message suivant pour lancer cette boucle externe:

```
(top read)
```

L'intérêt d'une telle représentation de la boucle externe, est de pouvoir disposer de l'expression littérale du message qui a été évalué au toplevel.

Cette structure de contrôle est particulièrement utile pour réaliser des systèmes de "dialogues entre experts" (Yonezawa.80), car il est possible de réaliser des formes qui s'appellent mutuellement sans utiliser la récursivité (c'est à dire sans saturer la pile), tout en sachant quel est l'émetteur du message.

De plus il est possible de se reconstruire des messages d'erreur de manière à ce que ceux ci puissent incorporer directement des techniques de Justification, Contestation et Silence (Berthelin.84), c'est à dire en réagissant par l'une des trois décisions suivantes: Inférence explicitée (justification), rejet avec amorce de dialogue (contestation) ou mise en attente (silence).

I.4 les Entités

L'unité de base de MERING est l'entité. Une entité est un objet informatique doué de propriétés de représentations à l'instar des structures de données, et de capacités d'exécutions, à l'image d'un ensemble de procédures.

L'aspect **statique** d'une entité est constitué d'un ensemble d'**attributs**, chaque attribut pouvant prendre une ou plusieurs valeurs à l'intérieur d'un moule spécifié par des contraintes qui limitent le domaine de validité de ces valeurs.

Ces valeurs peuvent être aussi bien des symboles terminaux (s-expressions LISP) que d'autres entités.

L'aspect **dynamique** d'une entité est caractérisé par l'ensemble des **transmissions** qui lui sont associées, **comportements** et **réflexes**. Les comportements sont des transmissions attachées à l'entité dans son entier, et activées lors de la réception d'un message par l'entité, dans le cas d'une méthode, ou par un couple (entité,attribut), dans le cas d'un applicateur; les réflexes sont des transmissions rapportées à un attribut particulier et déclenchées seulement lors de sa manipulation.

Définir une entité revient donc à décrire l'ensemble de ses attributs (contraintes, valeurs et réflexes) et de ses comportements.

II.4.1 Types et représentants

Il est souvent intéressant de pouvoir structurer les informations suivant des caractéristiques proches de la théorie des ensembles. A cet effet nous avons repris l'idée, maintenant classique en Intelligence Artificielle, de la différenciation des entités entre **type générique** (ou classe, ou prototype) et '**représentants**' (ou occurrence, ou "instance"). Généralement on estime que les types servent à définir de manière intensionnelle toute une classe d'objets, et que les représentants sont des réalisations particulières de ce type.

Toute entité est issue d'un type. C'est évident pour les représentants, mais les types génériques aussi sont dans cette situation en étant représentant du "méta-type" **type**. L'utilisation de ce méta-type permet de conserver sa cohérence formelle et son homogénéité au langage, en instituant que la méthode => (nouvelle entité) ait toujours pour fonction la création d'un représentant, et que toute opération ait lieu dans le cadre d'une transmission. (Les entités inconnues du système, c'est à dire qui n'ont pas encore été générées, appartiennent elles aussi à un type: le type **indéfini**).

Par exemple, si l'on veut décrire la classe des employés d'une entreprise, on construira un type générique signifiant qu'un employé est une personne qui a un numéro de sécurité sociale, un rôle au sein de l'entreprise (les symboles précédés du caractère \$ sont des mots réservés du langage), etc...

```
: (type => () employe
    secu $dom    num-secu,
    role $dom    role,
    statut $req  '(os op oq agent-maitrise cadre),
)
= EMPLOYE
```

Une personne particulière, Alfred, sera alors décrite comme un représentant de ce type. Le symbole <- décrit la facette (ou champ) contenant la valeur de l'attribut. Ce symbole est le

même que celui de l'applicateur d'affectation, bien que son sens soit différent.

```
: (personne => alfred
   secu  <- 1550775114005,
   role  <- comptable,
   statut <- agent-maitrise,
 )
= ALFRED
```

Ces définition sont elle mêmes des transmissions. La première est le message '=>' envoyé au méta-type 'type', et signifie "créer un représentant", c'est à dire dans le cas présent un type.

Le même message est envoyé dans le second cas, mais le receveur étant cette fois un type, il y a création d'un représentant: alfred.

Il y a donc trois catégories d'entité en MERING: les méta-types (catégorie réduite au seul élément type), les types qui en sont des occurrences, et les représentants des types.

La notion de méta-type en MERING, est moins formalisée que dans un langage comme SMALLTALK. Dans ce dernier chaque classe est rapportée à une méta-classe qui définit la sémantique de la classe (noms des champs et méthodes de classes). La distinction entre type et représentants est moins catégorique en MERING: un type répond aux mêmes messages que ses représentants, il dispose de valeurs propres, etc... De ce fait le méta-type type ne comprend que quelques méthodes générales, et surtout la méthode servant à la création de nouveaux types.

A ce sujet nous projetons, dans une version ultérieure de MERING de rajouter à nos comportements généraux, des comportements spécifiques aux représentants ou aux types.

En fait cette distinction entre types et représentants doit être assouplie et mieux comprise. En MERING ce découpage est un peu artificiel et sert surtout à gérer les "héritages" de propriétés, car les entités se comportent toutes de la même

manière qu'il s'agisse de types ou de représentants. En particulier il est possible de placer des valeurs dans l'attribut d'un type, comme de spécifier des comportements, des attributs ou de simples contraintes relatives à un représentant.

Ex:

```
: (employe => jules
  yeux-bleus $dom $bool ; l'attribut YEUX-BLEUS ne sera ;
  $default T, ; connu que de l'employe JULES ;
)
= JULES
```

En outre, toutes ces précisions peuvent être apportées de manière dynamique, c'est à dire ajoutées, transformées ou supprimées en cours d'exécution. Par exemple, si nous ne savons pas précisément quelle est le statut d'Alfred, mais que nous sachions qu'il est soit cadre, soit agent de maitrise, il serait possible de limiter le domaine de l'attribut 'statut' d'Alfred à ces deux valeurs seulement grâce à la transmission:

```
(alfred statut <-fac '$valeurs '(agent-maitrise cadre))
```

II.4.2 Héritage des propriétés

Il existe une hiérarchie entre les types génériques qui permet de limiter le nombre d'informations à préciser lors de la création d'un nouveau type, en utilisant un système d'héritage de propriétés, ce que Fahlman (Fahlman.79) appelle la copie virtuelle.

L'héritage vertical qui relie les représentants aux types, et les types entre eux à l'aide d'une hiérarchie, permet à une entité de disposer de **toutes** les caractéristiques d'une entité définie plus haut dans la hiérarchie, sauf si cette propriété a été redéfinie (comme dans le cas de langages comme SIMULA ou SMALLTALK). Par exemple il est possible de créer un type **cadre**, sous type de **employe**:

```

: (type => employe cadre
  statut <- cadre, ; le type CADRE est defini comme un employe ;
) ; qui a un statut de cadre ;
= CADRE

```

De cette manière toutes les entité définies comme représentant de **cadre** recevront **cadre** comme valeur de l'attribut **statut**.

Cet héritage est valable non seulement pour les attributs mais aussi pour les comportements. Une méthode définie pour une entité reste valable pour tous ses descendants tant qu'elle n'est pas redéfinie.

Ex: définition simplifiée du type BOX correspondant à la classe BOX de SMALLTALK (Kay.76) telle qu'elle a été reprise dans BJVLIISP (Cointe.82):

```

: (type => $objet box
  O $dom $nombre,
  LO $dom $nombre,
  LA $dom $nombre,
  ; (ce X) est une contraction syntaxique pour (% X ?) ;
  ; la fonction square dessine un carre sur le terminal ;
  ; elle est ecrite en LISP ;
  draw methode: () (square (ce O) (ce LO) (ce LA) & %),
  move methode: (NO) (% O <- (ce O + NO) & %),
  grow methode: (NL1 NL2) (% LO <- (ce LO + NL1)
    & % LA <- (ce LA + NL2) & %),
  turn methode: () (mlet ((aux cell))
    (aux <- LO
      & % LO <- (ce LA)
      & % LA <- aux) & %),
)
= BOX

: (box => toto | O <- 3 | LO <- 4 | LA <- 4)
= TOTO

: (box => riri | O <- (2 + 3) | LO <- (1 * 3) | LA <- 5)
= RIRI

```

```
: (toto draw)
```

```
  * * * *
  *      *
  *      *
  * * * *
```

```
= TOTO
```

```
: (riri draw)
```

```
  * * *
  *      *
  *      *
  *      *
  * * *
```

```
= RIRI
```

```
: (riri turn)
```

```
  * * * * *
  *          *
  * * * * *
```

```
= RIRI
```

```
: (toto move 10)
```

```
  * * * *
  *      *
  *      *
  * * * *
```

```
= TOTO
```

```
: (riri turn | move 5 | draw)
```

```
  * * *
  *      *
  *      *
  *      *
  * * *
```

```
= RIRI
```

Toute entité peut hériter des attributs et comportements de plusieurs branches différentes. On parle alors d'héritage multiple des propriétés. Par exemple, le type **voiture** est sous-type de **véhicule**. Mais une voiture c'est aussi un élément de standing. Il s'agit de points de vues différents du même concept. Toutes les voitures ont alors des caractéristiques issues à la fois des types **voiture** et **element-standing**.

Cette multiplicité des sources peut être réalisée au sein même du langage. Chaque type est issu d'un nombre quelconque de types plus généraux que lui, comme le montre la figure 2.4

Ex:

```
: (type => (vehicule element-standing) voiture)
= VOITURE
```

Cependant, afin d'éviter des collisions entre attributs ou comportement homonymes, une priorité a été établie entre les sources. Lorsqu'un conflit surgit entre plusieurs sélecteurs ayant le même nom, le système ne considère que le premier de la liste, ignorant parfaitement les autres: ce mécanisme introduit un ordre sur l'ensemble des types pères.

Cet ordre est dû à l'implantation du mécanisme d'héritage: celui ci effectue une exploration en profondeur d'abord sur l'ensemble des types pères pour retrouver l'attribut (et la facette particulière en jeu à un moment précis) ou le comportement en question.

D'autres techniques auraient pu être employées: utilisation d'identificateurs rappelant l'origine de l'héritage ou levée des interférences potentielles lors de la création des types (Borning.82). Nous avons préféré employer la méthode des priorités pour les raisons suivantes:

- 1- possibilité de masquage des noms. Pour supprimer l'héritage d'une propriété, il suffit d'employer le même nom pour une caractéristique appartenant à un type père disposant d'une priorité plus importante.
- 2- Non intervention du programmeur lors de la création des types, bien qu'un système qui lève les ambiguïtés par saisie interactive (d'ailleurs actuellement en cours de réalisation) soit très facile à mettre en oeuvre.
- 3- Facilité d'utilisation lors de création dynamique de types. Dans ce cas les techniques interactives sont inopérantes, et les solutions qui consistent à modifier le nom des sélecteurs s'avèrent beaucoup trop lourdes.

Il est cependant toujours possible de s'affranchir de cet ordre grâce à l'indicateur **super:**. Celui ci indique à partir de quel type les recherches concernant les sélecteurs doivent être effectuées.

La syntaxe de ce mécanisme est la suivante:

(**super:** sélecteur type)

dans lequel 'sélecteur' est le nom d'un comportement (méthode ou applicateur) ou d'un attribut, et 'type' est le nom du type à partir duquel les recherches doivent être effectuées. Si la spécification du type est manquante, l'origine de la recherche est calculée comme suit: si le receveur du message est un type alors, la recherche commencera sur le premier sur-type; en revanche, s'il s'agit d'un représentant, l'exploration débutera sur le premier grand-père.

L'utilisation de **super:** permet de résoudre des conflits entre sélecteur de même nom, d'utiliser dans la description d'un comportement une méthode définie dans un type situé au dessus ou tout simplement d'accélérer la recherche d'un attribut ou d'un comportement.

Ex:

```
: (type => $objet vehicule
  fonction <- transporter,
)
= VEHICULE

: (type => $objet element-standing
  fonction <- representation-sociale,
)
= ELEMENT-STANDING

; une voiture est à la fois un véhicule et un élément de standing ;
: (type => (vehicule element-standing) voiture )
= VOITURE

: (voiture fonction ?)
= TRANSPORTER ; une voiture est avant tout un véhicule ;
```

```

; mais il est toujours possible de s'en affranchir;
: (voiture (super: fonction element-standing) ?)
= REPRESENTATION-SOCIALE

; redéfinition de l'applicateur d'affectation <- pour qu'il ;
; retourne la valeur affectée et non le receveur du message ;

: (type => $objet truc
  <- applic: (V) (% # (super: <- ) V | V),
)
= TRUC

: (truc => foo
  A $dom $indef,
)
= FOO

: (foo A <- 'machin)
= MACHIN ; le nouvel applicateur ramene la valeur ;

: (foo A (super: <-) 'chose)
= FOO ; l'ancien applicateur retourne le receveur ;

```

4.3 Les comportements

Les comportements sont la partie réellement active des entités, constituant ainsi un véritable mode-d'emploi du type et ses représentants.

La hiérarchie entre les entités s'applique ici aussi: un représentant hérite des comportements déclarés dans le type dont est issu, lequel hérite lui-même des comportements définis dans les types situés au dessus.

Tout comportement possède un certain nombre de paramètres

formels qui seront substitués aux arguments réels accompagnant le message d'activation.

Il existe deux catégories de comportements:

- 1) Les méthodes qui décrivent les actions à entreprendre à la réception d'un message destiné à l'entité toute entière (une M-transmission)
- 2) Les applicateurs qui spécifient le comportement de l'entité après réception d'un message destiné à un attribut (une A-transmission)

Par exemple :

(toto print)

est la méthode 'print' envoyée à l'entité 'toto', et qui lui demande de s'imprimer, c'est à dire de révéler sa structure interne. Ce message est envoyé à l'entité toute entière et non à un attribut particulier.

En revanche,

(toto age <- 32)

est une transmission qui demande au couple (toto,age) de restituer sa valeur. Ici le message est adressé directement à l'attribut 'age' de l'entité 'toto'.

Cette distinction entre ces deux types de comportements est spécifique à MERING. Les méthodes MERING sont semblables aux méthodes SMALLTALK. En revanche, les applicateurs ont été introduits à cause de la notion de réflexe. Ils servent non seulement à manipuler les attributs mais aussi à programmer le déclenchement des réflexes. L'interrelation entre applicateurs et réflexes sera discutée plus en détail au chapitre II.4.4

MERING caractérise ses deux types de comportements en plaçant la marque **methode:** ou **methode|** lorsqu'il s'agit d'une méthode et **apporteur:** ou **apporteur|** dans le cas d'un applicateur. Le choix entre ces deux formes de la même marque indique si le comportement évalue ses arguments

ou non.

Un applicateur et une méthode peuvent porter le même nom sans entrer en conflit. L'évaluateur saura retrouver les siens lors de l'exécution.

De ce fait, il est possible de définir une méthode (un applicateur) à l'aide d'un applicateur (une méthode) qui porte le même nom. Par exemple un applicateur 'print' qui visualise la valeur d'un attribut pourrait être défini en appliquant la méthode 'print' à chacune des entités que comportent cet attribut (rappelons que les symboles % et # signifient "cette entité" et "cet attribut"):

```
print applicateur: () (forall X (% # ?) (X print)),
```

Il y a toujours priorité de l'applicateur sur la méthode. Voici quelques exemples paradoxaux (mais qui fonctionnent parfaitement) qui montrent comment MERING distingue entre attributs, méthodes et applicateurs.

- 1) (toto age <- 40)
- 2) (toto age 40)
- 3) (toto <- <- 40)
- 4) (toto <- 40)

Dans le premier cas MERING interprète cette transmission comme l'affectation de la valeur 40 à l'attribut 'age' de 'toto'. Il s'agit donc d'une A-transmission, et '<-' est considéré comme un applicateur.

Dans le deuxième exemple, il s'agit d'une M-transmission et 'age' est une méthode qui se définit comme suit:

```
age methode: (X) (% age <- X),
```

Le troisième exemple suppose qu'il existe un attribut '<-' auquel on affecte la valeur 40.

La quatrième expression est une M-transmission qui prend la

valeur 40 comme argument, et qui suppose que '<-' est ici une méthode.

Toutes ces formes peuvent se trouver réunies au sein de la même entité sans entrer en conflit:

```
(type => $objet exemple
  age      $dom      $nombre,
  age      methode: (X) (% age <- X),
  <-      $dom      $nombre,
  <-      methode: (X) (% <- <- X),
)
```

Description des comportements de base

MERING comprend un certain nombre de comportements de base, qui font partie du langage lui-même. Ils se trouvent dans les types `$sysroot` et `$sysobj`, et sont donc utilisables par toutes les entités de la base. D'autres sont plus spécifiques à des types représentant des structures de données tels que les types `cell` ou `seq`

Ces comportements se divisent en plusieurs catégories:

- methodes générales
- Tests de types
- Manipulation de facettes
- Appicateurs actifs
- Prédicats de correspondance
- Manipulation de listes (générales, indexées et associatives)

Une description de leur fonctionnement est proposé au

chapitre IV.

4.4 Les attributs

Les attributs supportent la partie purement formationnelle des entités, c'est à dire les relations qu'ils établissent avec d'autres objets, en opposition aux méthodes et applicateurs qui représentent l'aspect comportemental. Un attribut n'est pas réduit à sa valeur, mais comporte un certain nombre de facettes qui le décrivent sous ses différents aspects.

La notion d'attribut doit être rattachée à celle des objets, comme les appellent les Anglo-saxons, éléments constitutifs des 'frames'. Comme ceux-ci les attributs sont définis aussi bien à l'aide d'informations déclaratives (contraintes, valeurs par défaut, etc..) que par des données procédurales (réflexes).

4.4.1 Modèle général

Au fur et à mesure du développement de MERING, notre compréhension des attributs s'est améliorée: au début l'attribut était qu'un ensemble d'informations dont nous pouvions parler, et que nous utilisions de manière pragmatique. Cette vision aurait pu suffire. Mais l'introduction de la notion d'applicateur lié à celle de réflexe nous a conduit à une tentative de formalisation que voici:

Un attribut ne doit pas être confondu avec les

informations (généralement des entités) qu'il contient: c'est pourtant ce qui se passe dans les langages orientés objets tels que SMALLTALK où l'on parle non pas d'attributs mais de l'ensemble des champs locaux, ou variables rémanentes qui sont associées à l'objet. Dans ce cas la liste des attributs se résout à un environnement de la forme:

$$((nom_1 . val_1) \dots (nom_n . val_n))$$

c'est à dire à une liste de doublets $(nom_i . val_i)$ où nom_i est le nom du ième champ et val_i sa valeur (une entité ou toute autre S-expression).

Un attribut en revanche doit être considéré comme une **entité locale** qui contient un certain nombre de champs (qui récursivement pourraient être envisagés comme d'autres attributs, etc..) et un certain nombre de méthodes (et des applicateurs si l'on poursuit récursivement en disant qu'il existe des attributs d'attributs) qui lui sont rattachées et que nous nommons les **réflexes**.

En d'autres termes, l'environnement local associé à une entité est de la forme:

$$((nom_1 \text{ attr}_1) \dots (nom_n \text{ attr}_n))$$

où nom_i est toujours le nom de l'attribut, mais où $attr_i$ n'est plus une variable mais un représentant du type **attribut** qui se définit ainsi:

```
(type => $sysobj attribut
  context $dom $sysobj, ; le nom de l'entité qui contient l'attribut
  nom      $dom $indef, ; le nom de l'attribut dans l'entité{
  $val     $dom $indef, ; la facette $val est celle qui contient
  $dom     $dom $indef, ; la valeur de l'attribut, elle se nomme
  $default $dom $indef, ; aussi <- ou ::
  $valeurs $dom $indef, ;
  $ordre-lect $dom $indef
      $valeurs ($default $si-req $regle-req - +),

  $req      methode: (##) (), ; ces méthodes sont définies au niveau;
  $si-req   methode: () (), ; de chaque attribut particulier ;
  $si-ajout methode: (##) (),
  $si-err-ajout methode: (##) (" ** erreur ajout valeur dans :\" print
                                & (ce context) print
                                & send: erreur)

  etc...
)
```

On constate donc que le modèle initial s'est élargi: il existe une indirection supplémentaire entre l'entité receveuse et la valeur de l'un de ses attributs, laquelle indirection est gérée par l'attribut lui-même et l'ensemble des applicateurs que connaît l'entité.

Les réflexes sont alors des méthodes spécifiques qui doivent être redéfinies pour chaque attribut particulier. Le type attribut contient seulement la description par défaut de ces méthodes.

Ces méthodes sont déclenchées par envoi de messages de manière très classique. Par exemple, déclencher le réflexe **si-ajout** de l'attribut A12 s'exprime:

```
(A12 $si-ajout)
```

En supposant que l'attribut A12 est connu par l'entité **toto**, sous le nom **etat**, déclencher ce réflexe revient à évaluer la transmission suivante:

```
(toto ? 'etat ù $si-ajout)
```

où la méthode ? retourne l'attribut A12

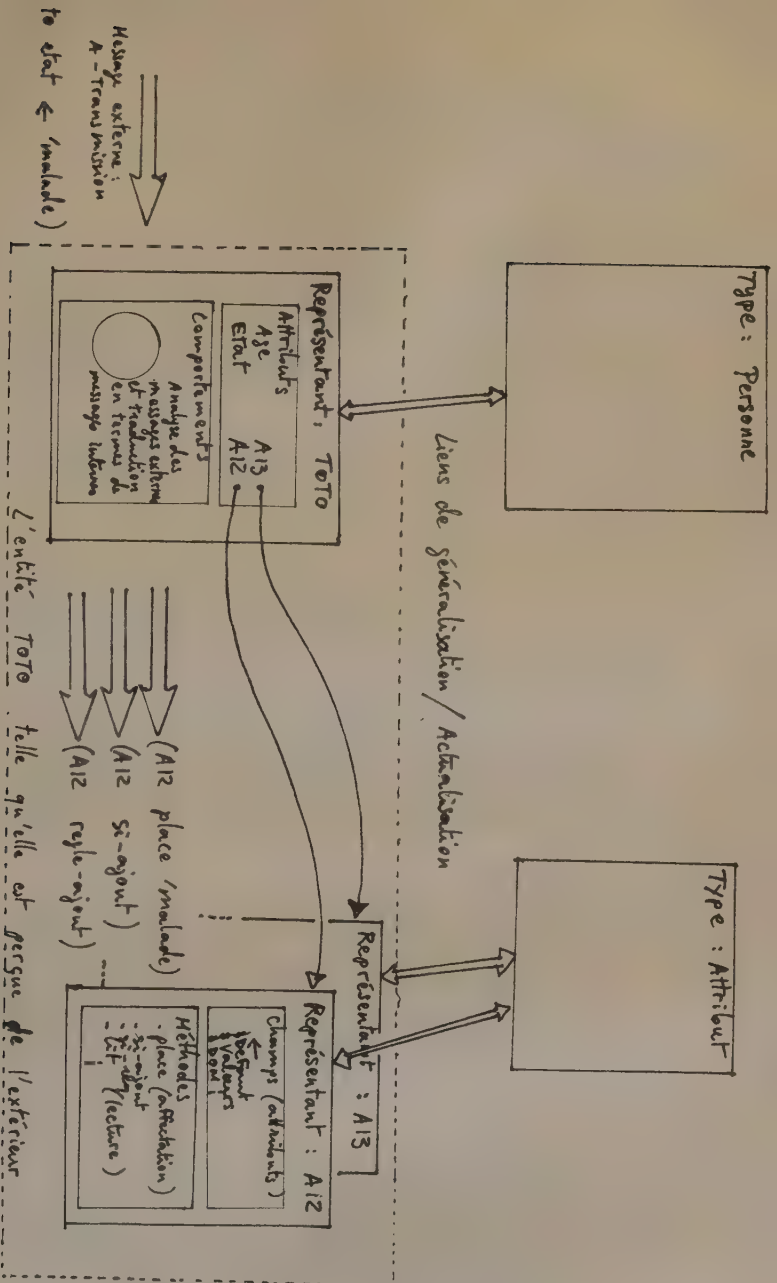


Figure 22 Structure Jernella des attributs ou MFKINE

Dans ce cadre les applicateurs sont la liaison entre le monde extérieur de l'entité et ses attributs en redéfinissant et redistribuant les messages provenant du monde extérieur pour qu'ils soient compris par l'attribut. La figure 2.2 schématise ce modèle en représentant l'ensemble des applicateurs connus par une entité comme un centre de redistribution des messages destinés aux attributs de l'entité.

Les facettes sont alors les différents attributs et comportements d'un attribut.

II.4.4.2 Les réflexes: des mécanismes d'inférence

Nous avons développé un système d'inférence qui à la fois généralise dans ses possibilités et simplifié dans son implantation, un mécanisme d'inférences bien connu en Intelligence Artificielle: les **démons**. Mais à l'opposé des démons, les différentes catégories de réflexes ne sont pas fixées par le système, mais redéfinissables par l'utilisateur.

En effet, à l'origine, la notion de "démon" faisait référence à un petit personnage virtuel qui passe son temps à observer ce qui se passe dans l'univers, et qui entre en action lorsqu'un événement particulier, pour lequel il a été dressé, se produit. Il s'agit donc essentiellement d'un mécanisme de scrutation en parallèle du flot normal de processus de calcul. L'implantation d'une telle technique est couteuse en temps (sur un processeur unique, une partie du temps de calcul est passé à gérer les démons) et en effort (il faut prévoir un mécanisme pour traiter ce parallélisme).

Par la suite la notion de démon s'est modifiée: en KRL, un démon est une procédure déclenchée lors de l'ajout d'une valeur à un attribut: pour pallier cette restriction les auteurs de ce langage ont ajouté les "servants", procédures activées lorsqu'un objet requiert des informations. En LOOP, les démons et les servants ont fusionnés en une seule notion: les "valeurs actives".

La notion de réflexe est à la fois plus simple et moins couteuse que celle des démons dans leur conception originelle, mais aussi plus puissante que les démons, servants et valeurs actives de KRL et LOOPS.

Un réflexe est directement déclenché par évaluation d'un "applicateur". Il n'y a donc pas de propriété posée a priori sur la nature des réflexes. Tel réflexe sera déclenché parce que l'attribut associé a reçu un message particulier: il peut s'agir de lecture, d'ajout ou de n'importe quel autre message envoyé à cet attribut. L'applicateur se bornant à redéfinir et redistribuer le message de l'entité vers l'attribut concerné.

Les applicateurs sont donc les "programmeurs" des réflexes: par exemple, en MERING deux réflexes peuvent être activés lors de la lecture d'un attribut: **\$si-req** et **\$si-lu**.

Le premier correspond directement à la notion de servant, c'est à dire qu'il est activé si aucune valeur n'est associée à l'attribut.

En revanche, le second, s'il est présent (c'est à dire si un réflexe '**\$si-lu**' a été associé à l'attribut en question) est toujours déclenché, court circuitant les autres mécanismes de lecture.

L'applicateur de lecture '??' déclenche le réflexe **\$si-lu**, alors que l'autre applicateur de lecture '??' inhibe ce dernier tout en conservant les mécanismes de valeurs par défaut, réflexe **\$si-req**, etc..

```
: (type => $objet foo
  A      *dom $nombre,
        $si-req (2 + 3),
  B      $dom $nombre
        <- 4
        $si-lu (3 + 5),
)
= FOO
```

```

: (foo => fool) ; création d'un représentant du type F00 ;
= F001

: (fool A ?) ; l'attribut A ne possède pas de valeur
associée ;
= 5 ; valeur ramenée par le réflexe $SI-REQ ;

; affectation d'une valeur puis relecture de l'attribut ;
: (fool A <- 7 & fool A ?)
= 7 ; valeur associée à l'attribut A ;

: (fool B ?)
= 8 ; valeur ramenée par le réflexe $SI-LU ;

: (fool B ??)
= 4 ; valeur associée à l'attribut B ;

```

Grâce à ces deux applicateurs qui déclenchent des exes différents, il est possible de créer des "héritages zontaux" (Bundy.79)(Fox.79), c'est à dire des héritages de riétés qui ne suivent pas le mécanisme d'héritage par liens te-de" ou "isa". Par exemple, l'héritage par transitivity amis des amis sont des amis) est caractéristique d'un tage horizontal. En MERING, la transitivity sur les parties objet (les parties des parties sont des parties) s'écrit i:

```

: (type => $objet objet
  parties $dom objet
    $si-lu (% parties ??
      | union (forall X (% parties ??)
        \ (X parties ???)),
    )
= OBJET

: (type => objet voiture
  parties <- (roue moteur),
  )
= VOITURE

: (type => objet roue
  parties <- (jante pneu),
  )
= ROUE

```

```

: (type => objet moteur
  parties <- (piston culasse bielle),
)
= MOTEUR

: (voiture parties ?)
= (ROUE MOTEUR JANTE PNEU PISTON CULASSE BIELLE)

```

Dans ce cas, ce qui est intéressant, ce n'est pas que l'on ait un mécanisme de transitivité (la plupart des transitivités du genre "les parties des parties sont des parties" ou "les amis des amis sont des amis" cessent d'être pertinentes au bout d'un certain temps: ces transitivités ne fonctionnent que dans un environnement local) mais qu'on puisse le définir sans aucune difficulté, et l'adapter aux besoins au fur et à mesure du développement d'une application.

D'autre part il est intéressant d'ordonner les différents types d'inférences que l'on peut utiliser lorsqu'on a besoin de la valeur d'un attribut et que celui ci est vide. S'il existe à la fois une valeur par défaut et une inférence de type 'si-req', doit on prendre a priori la valeur par défaut ou celle qui est retournée par la transmission? Il est nécessaire à la fois d'imposer un ordre sur ces opérations et d'être aussi en mesure de le modifier.

Pour toutes ces raisons nous avons supprimé les notions de démons et de servants au profit de celle de réflexe.

Un **réflexe** est une opération automatique entreprise par une entité lors de la manipulation d'un de ses attributs. La manipulation activatrice n'est plus définie a priori mais dépend de la description de l'opérateur responsable de la manipulation, c'est à dire de l'applicateur.

L'ensemble des informations nécessaires au bon déroulement de ces réflexes (transmissions, valeurs, données diverses) est placé dans des facettes associées à chaque attribut (c'est à dire dans les attributs de l'attribut), et qui bénéficient de l'héritage vertical.

De ce fait les applicateurs et les réflexes sont étroitement liés au sein de couples (applicateur, réflexes) qui définissent quand et en fonction de quel critères les réflexes doivent être déclenchés.

MERING dispose à l'origine d'un certain nombre de réflexes, mais la création de nouveaux mécanismes d'inférence (et nous ne parlons pas seulement du contenu de ces inférences mais aussi de leurs propriétés structurelles: quand et à quelles occasions vont elles se déclencher, quel va être leur effet sur la suite normale des opérations, etc...) est mise à la disposition et à la portée du programmeur. Pour créer un nouveau type d'inférence en MERING, il suffit de distinguer de nouvelles facettes pour supporter ces informations et de décrire l'applicateur correspondant dans le langage lui-même.

II.4.4.3 Les facettes

En MERING, l'ensemble des facettes pré-définies se divise en quatre groupes: **valeurs, filtres, réflexes et utilitaires.**

Les premières ont pour objet de recevoir la (ou les) valeur(s) que peut prendre un attribut, ainsi que les valeurs par défaut, c'est à dire les informations implicites qui caractérisent le prototype d'une entité.

Les secondes agissent comme des contraintes sur l'ensemble des valeurs permises pour un attribut. Il s'agit des domaines de définition, caractérisés aussi bien en extension qu'en intension. De plus ces contraintes sont aussi des informations qui sont utiles pour savoir la valeur que porte un attribut quand celle ci n'a pas été précisée. Par exemple le domaine de définition de l'attribut 'agent' d'une action telle que vendre, est 'personne'.

```
(type => $objet vendre
  agent $dom personne,
  destin $dom personne,
  theme $dom chose,
)
```

Ainsi, même sans savoir "qui vend des chaussettes", il est possible de dire que c'est une personne.

Les réflexes sont la partie active des attributs, des transmissions activées lors de la manipulation d'un attribut. En MERING, les réflexes généralisent la notion de démons et de servants de KRL, en permettant à l'utilisateur de créer ses propres catégories de réflexes, en association avec les applicateurs qui les déclenchent.

Les facettes utilitaires contiennent des informations d'ordre générales qui peuvent être utiles lors de la définition des entités et qui n'entrent pas dans les différents types de facettes pré-citées.

Toutes les données qui contiennent ces facettes sont transmises par héritage vertical, c'est à dire qu'une véritable copie virtuelle existe aussi bien de types à représentants que de types à sous-types, pour les informations contenues dans les facettes.

Le contenu de toutes ces facettes est modifiable dynamiquement, à l'aide d'applicateurs destinés à cet effet. De plus leur nombre n'est pas fixé à l'avance, de nouvelles facettes pouvant toujours être redéfinies.

C'est pourquoi nous ne présentons ci-dessous que les facettes de base. Nous verrons par la suite comment intégrer de nouvelles caractéristiques à MERING par la création de facettes et d'applicateurs adéquats.

Facettes valeurs

Facette **\$val** : Cette facette contient la valeur que prend l'attribut lors de sa réalisation explicite. Cette facette est représentée par le signe '<-'.

```
: (jules profession <- journaliste)
= JULES
```

```
: (jules print)
(PERSONNE => JULES
  PROFESSION <- JOURNALISTE
)
= JULES
```

Facette **\$default** : Il est possible d'associer une (ou plusieurs) valeur(s) par défaut à un attribut, en plaçant cette information dans la facette '\$default'. Ainsi, à la lecture d'un attribut, cette valeur sera employée à la place de la valeur réelle.

```
: (personne print)
(TYPE => CONCEPT PERSONNE
  REMUNERATION $DOM REMUN-TYPE
  $DEFAULT SALARIE,
)
= PERSONNE
```

```
: (jules remuneration ?)
= SALARIE
```

Les facettes filtres

- Facette **\$dom** : Cette facette décrit l'ensemble des types auxquels peut appartenir la (ou les) valeur(s) de l'attribut. Cette facette doit exister pour chaque attribut, même si la valeur n'a pas d'importance. Dans ce cas on place l'indication **\$any** qui supprime ce test lors de l'affectation ou de l'ajout d'une valeur à

l'attribut en question. Les types \$nombre et \$chaine se rapportent respectivement à un nombre ou à une chaîne de caractères.

- Facette \$num : indique le nombre d'éléments dont la facette \$val contient l'unité. Par exemple si l'on dit qu'un chat a quatre pattes, la valeur 4 sera placée dans la facette \$num de l'attribut 'membre'.
- Facette \$req : est la partie active. Cette facette contient une transmission qui sera évaluée lors de l'affectation (ou de l'ajout) de l'attribut. Si la valeur introduite ne convient pas au filtre, la transmission située sur la facette '\$si-err-ajout' est activée. (cette valeur testée est représentée par le symbole ff).
- Facette \$valeurs : Lorsque la valeur n'appartient pas à un domaine précis (indication '\$any' dans la facette \$dom) il est cependant possible de préciser de manière exhaustive les différentes valeurs permises grâce à cette facette.


```

      : (type => $objet objet
        couleur $valeurs (rouge bleu noir blanc
          vert orange),
        )
      = OBJET
      
```
- Facette \$mode : caractérise le nombre de valeurs que peut prendre un attribut. Il existe deux modes possibles: univalué et multivalué (représentés par les mots clés \$uni et \$multi. Dans le premier cas, la valeur se réduit impérativement à une seule entité ou à un seul atome. Cependant dans le cas général, il peut être intéressant d'associer une liste d'éléments à un attribut, le mode devient alors 'multivalué'.

L'utilisation d'un attribut appartenant à un mode ou à un autre entraîne des contraintes sur des applications telles que l'ajout d'une valeur. En effet, s'il est univalué, l'attribut ne peut supporter deux valeurs. L'introduction d'une valeur supplémentaire entraîne le déclenchement de la transmission située sur la facette 'si-err-mode'. Par défaut un attribut est considéré comme multivalué, les contraintes quant aux nombres de valeurs étant supprimées.

Ex:

Les employés d'une entreprise pourraient être décrits à l'aide d'un type générique 'employe', sous-type de personne, qui prend en compte aussi bien leurs qualités statutaires que les différentes fonctions qu'ils ont au sein de l'entreprise:

```
: (type => personne employe
  entreprise $dom entreprise
              $mode $uni,
  statut     $dom $indef
              $valeurs (os op oq agent-maitrise cadre)
              $mode $uni,
  emploi     $typ-emploi ; appartient a la categorie
de l'entreprise;
  $req (ce entreprise categorie ? |
typ-emploi == ##),
  salaire    $nombre
              $req (## > (ce emploi remun-minimum ?))
              $mode $uni,
  fonction   $dom travail,
)
= EMPLOYE
```

Facettes réflexes

Toutes ces facettes contiennent des messages, les réflexes, qui sont activés lors de la réception d'une A-transmission: affectation, ajout, lecture et suppressions de valeurs notamment.

- Facette \$si-req : Lorsque l'on désire lire un attribut (applicateur ?) et que celui ci ne comporte pas de valeur, le réflexe que contient cette facette est déclenché. Il s'agit là d'une technique bien connue sous le nom de servant (KRL), ou de procédures 'IF-NEEDED' (FRL). A la différence de ces langages, il ne s'agit pas de procédures, c'est à dire d'éléments informatiques n'appartenant pas au système de représentation, mais de transmissions écrites en MERING. Par exemple, il est possible de définir une entité au cours de l'activation d'un réflexe:

```

: (type => $objet foo
  A      $dom foo
        $si-req (foo => )
)
= F00

: (foo => fu | A ?)
= F002

: (fu print)
  (F00 => FU
    A <- (F002)
  )
= FU

```

- Facette \$si-lu : Le réflexe attaché à cette facette est toujours déclenché à la lecture de l'attribut si celui-ci possède une valeur. Il s'avère particulièrement intéressant pour construire des héritages horizontaux comme nous

venons de le voir.

Il court circuite le fonctionnement normal de l'applicateur de lecture. Il est de ce fait possible de lancer toute une transmission complexe à la lecture d'une valeur de manière à réaliser des systèmes de trace automatique par exemple. Il est possible cependant de supprimer l'action de ce réflexe en employant l'applicateur '??' au lieu de '?'

- Facette \$si-ajout : Cette facette contient une transmission qui est déclenchée lors de l'affectation ou de l'ajout d'une valeur à l'attribut (applicateurs <- et <=). Son fonctionnement est à rapprocher des démons de KRL et de la facette IF-ADDED de FRL. Cette transmission utilise un symbole particulier ##, qui représente la valeur placée dans l'attribut. Ce réflexe n'est activé que si la valeur franchit les filtres et qu'il s'agit donc d'une donnée valable.

```

: (type => personne capitaine
  bateau $dom bateau,
  age    $dom $nombre
        $req (## > 0)
  $si-ajout (" l'age du capitaine est "
prinl
                                & ## `print),
)
= CAPITAINE

: (capitaine => barbe-noire)
= BARBE-NOIRE

: (barbe noire age <- -5)
  "** ERREUR AJOUT VALEUR DANS " BARBE-NOIRE

: (barbe-noire age ?)
= NIL

: (barbe-noire age <- 54 | age ?)
  "l'age du capitaine est " 54
= 54

```

- Facette **\$si-suppr** : Le réflexe associé est déclenché si une valeur de l'attribut est supprimée. Le symbole **##** représente alors la valeur supprimée:

```
: (capitaine age <-fac '$si-suppr
                                '("le capitaine n'a plus l'age"
prinl & ## print))
= CAPITAINE

: (barbe-noire age del)
"le capitaine n'a plus l'age" 54
= 54
```

- Facette **\$regles-req** : contient les noms des règles qui doivent être activées lors d'une requête de lecture et qu'aucune valeur n'est présente dans la facette **\$val**. Il s'agit donc d'un réflexe déclenché à la place ou en superposition au réflexe '**\$si-req**'. (voir le chapitre III pour des exemples d'utilisation de cette facette)
- Facette **\$regles-lu** : Identique à la précédente, cette facette est déclenchée lorsqu'une valeur non nulle est présente dans l'attribut, lors d'une opération de lecture, après activation de l'éventuel réflexe '**\$si-lu**'. (voir le chapitre III pour des exemples d'utilisation de cette facette).
- Facette **\$regles-ajout** : contient les noms des règles qui doivent être utilisées lors de l'ajout d'une valeur à l'attribut. Cette opération est effectuée après déclenchement du réflexe '**\$si-ajout**'. (voir le chapitre III pour des exemples d'utilisation de cette facette).
- Facette **\$si-err-ajout** : Cette facette est activée lorsqu'on tente d'affecter ou d'ajouter une valeur invalide, c'est à dire rejetée par les

filtres, à l'attribut. L'icône ## symbolise cette valeur, et %% représente l'entité émettrice du message. Grâce à ces deux informations, il est possible de créer un système qui, lorsqu'une donnée invalide est envoyée, demande à l'émetteur de se justifier.

- Facette **\$si-err-mode** : contient la transmission qui est activée si l'on tente d'ajouter une donnée supplémentaire (ou d'affecter une liste de données) à un attribut univalué.

Facettes utilitaires

- Facette **\$ordre-lect** : Cette facette décrit l'ordre de déclenchement des transmissions, lors d'une lecture réalisée sans que l'attribut ne possède lui-même de valeur. Cette facette régit donc la priorité d'action entre les facettes '\$default', '\$si-req' et '\$règles-req'. D'autre part elle indique si '\$regle-req' doit ou non être déclenché même si '\$si-req' a déjà ramené une valeur.

L'utilisation de cette facette permet de modifier le comportement des réflexes et ainsi disposer d'une latitude plus grande dans les mécanismes de contrôle des définitions implicites. Il est souvent intéressant de pouvoir déclencher la facette '\$si-req' avant d'utiliser la valeur par défaut. Dans ce cas il suffit d'indiquer dans cette facette l'ordre '(\$i-req \$default)'. Le sens général s'il n'est pas précisé correspond à la liste:

(\$default \$si-req \$regle-req -)

Le signe - signifie que si la facette '\$si-req' ramène un résultat, la facette '\$regle-req' ne doit pas être déclenchée. En revanche le signe

+ indique que cette dernière devra être activée quel que soit le résultat des valeurs par défaut ou de l'évaluation de la facette '\$si-req'.

- Facette **\$question** : contient un message qui pourra être affiché lors de l'utilisation de la méthode 'questionner'. Cette facette est très pratique pour réaliser simplement des systèmes où le dialogue avec l'utilisateur est simplifié(ex: systèmes experts, questionnaire intelligent, réservations de places, etc...)

II.4.5 Définition des entités

Cette dernière section du chapitre II va nous permettre de résumer un certain nombre des points qui ont été abordés dans ce chapitre, en examinant comment sont définies les entités, qu'il s'agisse de types ou de représentants.

II.4.5.1 Construction des types et des représentants

Il est possible de construire une nouvelle entité à chaque moment, étendant ainsi l'univers MERING. Nous avons vu que toute expression évaluable prend la forme d'une transmission. Ceci est notamment vrai pour la création d'un type ou d'une entité en général.

Un type est considéré comme un représentant du méta-type 'type'. De ce fait, la création d'un type revient à l'envoi du message '=>' (nouveau représentant) à l'entité 'type', suivit des paramètres servant à caractériser ce type.

La syntaxe d'une définition de type est donc la suivante:

type => nom-des-sur-types nom-du-type corps-de-définition)

définition d'un type inclut

la liste des sur-types qui définit l'ensemble des types génériques dont le type en question est un sous-type. Leurs propriétés seront transmises par héritage vertical au type créé et à tous ses représentants. Lorsqu'aucune précision n'est apportée (c'est à dire si la liste est vide), on suppose qu'il s'agit par défaut du type \$objet.

Lorsque la liste est réduite à un seul élément, celle-ci peut alors prendre la forme d'un atome, un symbole unique qui représente l'unique type père.

L'ordre des types à l'intérieur de cette liste n'est pas indifférent. En effet l'héritage des propriétés est réglée, en cas de conflit entre noms locaux, par une priorité simple, introduite par l'ordre sur les types pères. Le type le plus à gauche est le plus prioritaire, le plus à droite étant choisi en dernier.

le nom du type qui doit obligatoirement être présent

le corps de la définition qui contient la sémantique du type, c'est à dire l'ensemble des attributs et des comportements qui décrivent le type.

Un représentant est créé au cours d'une transmission globale à celle qui caractérise la définition d'un type, et la syntaxe est la suivante:

nom-du-type => nom-du-représentant corps-de-définition)

les restrictions suivantes:

le nom du type doit obligatoirement être présent, et correspondre à un type déjà défini dans la base.

Le nom du représentant peut être absent. Dans ce cas,

un nouveau nom est engendré en concaténant le nom du type et un nombre, de manière à ce que ce nom soit nouveau et que la création de ce représentant ne vienne pas écraser une entité déjà existante.

- 3) Le corps de la définition décrit l'ensemble des propriétés spécifiques à ce représentant et qui le distingue du type: initialisation des valeurs des attributs, mais aussi des description de contraintes, de réflexes, de méthodes et d'applicateurs qui ne seront connus que de cette entité.

Qu'il s'agisse d'un type ou d'un représentant, la description du corps de la définition suit une certain nombre de principes:

- 1) Il n'y a pas d'ordre dans la description de ces propriété. Qu'une méthode apparaissent avant ou après un attribut n'a pas d'importance.
- 2) Chaque propriété (qu'il s'agisse d'un comportement ou d'un attribut) voit sa description terminée par une virgule (,).
- 3) Après définition du type, la méthode **newtype** est automatiquement lancée. Après définition d'un représentant, c'est la méthode **newrep** qui est activée. Ces méthodes n'existent que pour être redéfinies par l'utilisateur et lui permettre de décrire des actions qui seront effectuées au moment de la création d'une entité (phase d'initialisation).

II.4.5.2 Description des méthodes

La description d'une méthode suit l'un des deux formats suivants:

- i) nom-méthode **methode**: (liste-arguments) corps-méthode,
ou
- ii) nom-méthode **methode** | (argument) corps-méthode,

Dans le premier cas, la méthode évalue ses arguments. Dans le second elle estime que tout le reste de la transmission est une liste qu'elle passe sans évaluer.

Le corps de la méthode est une transmission dans laquelle apparaît des variables, c'est à dire des symboles définis dans la liste des arguments, et qui seront substitués aux objets réels lors de l'activation de cette méthode, à la réception d'un message.

Dans ces transmissions, le symbole % représente l'entité, non encore connue qui recevra le message. Cette formulation est à rapprocher des symboles SELF en SMALLTALK et THIS en SIMULA. Grâce à cette particularité, il est possible d'effectuer ce que l'on appelle des "self-transmissions"; c'est à dire d'envoyer des messages à l'entité elle même.

II.4.5.3 Description des applicateurs

La description d'un applicateur est très semblable à celle d'une méthode. Elle n'en diffère que par son étiquette (applic au lieu de methode), et par son icône #. La syntaxe est donc:

- i) nom-applicateur **applic:** (liste-arguments) corps-méthode,
ou
- ii) nom-applicateur **applic|** (argument) corps-méthode,

Selon que l'applicateur évalue ses arguments ou non.

Le symbole # représente l'attribut auquel le message qui active cet applicateur est destiné. A la réception d'un message, l'attribut et l'entité récepteurs sont donc substitués aux symboles # et % respectivement.

Description d'un attribut

Les attributs sont des extensions de la notion du couple (propriété,valeur). Ils sont constitués de facettes qui décrivent leurs différents aspects. La syntaxe de définition de ces attributs est la suivante:

```
nom-attribut  facette1 donnée1
               facette2 donnée2
               .
               .
               facetten donnéen,
```

Les données peuvent prendre la forme d'une entité (ou d'un atome), d'une liste d'entités ou encore d'une transmission.

La facette contenant la valeur propre de l'attribut, s'écrit <-. (Cependant lors de la définition d'une entité, il est possible de représenter cette facette par le symbole ::. Une facilité qui fait partie du sucre syntaxique.)

CHAPITRE III

Applications et extensions

Nous présentons dans ce chapitre un ensemble d'applications et d'extensions de MERING. Ces exemples sont destinés à montrer les capacités du langage dans des domaines divers: traitement de données, questionnaire intelligent, modélisation de problèmes par contraintes, mondes artificiels et analyse du langage naturel.

Ces petits programmes ne doivent pas être regardés comme des applications complètes (il n'y a pas de recherche à effectuer pour parler dans ces applications) mais servent à mettre en lumière les qualités inhérentes de MERING, sa simplicité de mise en oeuvre, sa consistance et ses capacités d'adaptation à des problèmes variés.

De plus, nous proposons quelques extensions au langage: la notion de 'classe' et la programmation par règles de production. Ces extensions sont destinées d'une part à montrer les qualités inhérentes de MERING, et d'autre part à offrir à l'utilisateur un environnement de programmation aussi riche que possible. De même, les "classes" MERING répondent à une préoccupation fondamentale en représentation des connaissances.

III.1 Base de données

Afin d'illustrer les capacités de MERING dans un domaine simple, nous avons réalisé une petite base de donnée portant sur le domaine journalistique. Trois types sont considérés: les personnes, les revues et les articles.

Il ne s'agit pas ici de rivaliser avec de réelles bases de données, mais de montrer comment il est possible de construire une base sur un domaine particulier. En effet, par sa formulation des connaissances sous formes d'entités, MERING se prête aisément à des applications simples de ce type.

Chaque type fournit les informations indispensables à l'établissement des entités de la base: qualités spécifiques, relations entre entités etc.

Les questions sont formulées à l'aide de transmissions MERING, qui se rapprochent beaucoup des langages d'interrogation utilisés dans les domaines de base de données. Par exemple une question telle que "quelles sont les revues qui publient à plus de 80 000 exemplaires" se formule ainsi:

```
(forall x (revues allinst)
  (if (x tirage > 80000)
    (x titre ? )))
```

ce qui signifie: ramener tous les titres des revues qui ont un tirage supérieur à 80 000.

Des prédicats ont été incorporés sous la forme de comportements, afin de simplifier la formulation des questions: par exemple: "dans quelle revues travaille alice" se réduit à la question:

```
(revue travaille? alice)
```

Voici la base de donnée complète:

```
(type => $objet journaliste
; les attributs des entites 'journaliste' ;
  nom $dom $indef,
  age $dom $nombre,
  revue $dom revue,
  statut $valeur (salarie pigiste),
; pour trouver les articles d'une personne ;
; il suffit de parcourir tous les articles ;
; en regardant leur auteur ;
  articles $dom articles
    $si-req (forall X (article allinst)
      (if (X auteur = %)
        X)),

; quelques utilitaires pour simplifier les requetes ;
; pour obtenir la liste des employes d'une revue ;
; si le message est envoye au type 'personne' ;
; sinon il indique si la personne travaille bien ;
; dans la revue ;
  employes methode: (R) (forall X (% allinst)
    (if (X revue == R)
      X)),

; pour obtenir l'ensemble des pigistes (message ;
; envoye au type) ou savoir si l'individu est ;
; pigiste. ;
  pigiste methode: (V) (exist X (% allinst)
    (if (X statut = pigiste)
      T)),

; pour savoir sous quelle reference une personne a ;
; ete enregistree ;
  ref? methode: (V) (exist X (personne allinst)
    (if (X nom = V)
      X)),
)

(type => $objet revue
; en premier: les attributs;
  titre $dom $indef,
  tirage $dom $nombre,
  periodicite $valeur (quotidien hebdo
    mensuel bimestriel trimestriel),
```



```

; pour trouver les journalistes, il suffit de regarder les rev
journalistes $dom journaliste
    $si-req (forall X (journaliste allinst)
        (if (X revue == %)
            X)),

; les methodes utilitaires ;
; pour trouver les revues a gros tirage ou savoir si ;
; une revue particuliere est a gros tirage ;
gros-tirage methode: () (% tirage > 70 000),

; pour obtenir tous les articles d'une revue ;
articles methode: () (forall X (% journalistes ?)
    (X articles ?)),

; pour savoir toutes les revues dans lesquelles un ;
; journaliste travaille ;
travaille? methode: (Journ) (forall R (% revue)
    (if (R journalistes == Journ)
        R)),
)

(type => $objet article
; le type article est tres simple ;
titre $dom $chaine,
auteur $dom journaliste,
prix $dom $nombre,
)

(journaliste => ()
nom <- alice,
age <- 33,
revue <- (micro-systemes monde-informatique),
statut <- pigiste,
)

(journaliste => ()
nom <- tintin,
age <- 24,
revue <- ordin-indiv,
statut <- salarie,
)

```

```
(journaliste => ()  
  nom <- amelie,  
  age <- 37,  
  revue <- (ordin-indiv ordin-poche),  
  statut <- salarie,  
)
```

```
(journaliste => ()  
  nom <- alfred,  
  age <- 32,  
  revue <- (telesoft ordin-indiv),  
  statut <- pigiste,  
)
```

```
(revue => ()  
  titre <- micro-systemes,  
  tirage <- 100 000,  
  periodicite <- mensuel,  
)
```

```
(revue => ()  
  titre <- monde-informatique,  
  tirage <- 40 000,  
  periodicite <- hebdo,  
)
```

```
(revue => ()  
  titre <- ordin-individuel,  
  tirage <- 80 000,  
  periodicite <- mensuel,  
)
```

```
(revue => ()  
  titre <- telesoft,  
  tirage <- 60 000,  
  periodicite <- bimestriel,  
)
```

```
(article => ()  
  titre <- "banc d'essai du ZX 81",  
  auteur <- alfred,  
  prix <- 400,
```

```
)

(article => ()
  titre <- "l'informatique a l'ecole",
  auteur <- alice,
  prix <- 900,
)

(article => ()
  titre <- "les memoires mortes",
  auteur <- tintin,
  prix <- 700,
)

(article => ()
  titre <- "un ordinateur en vacances",
  auteur <- alice,
  prix <- 800,
)

(article => ()
  titre <- "les memoires vives",
  auteur <- tintin,
  prix <- 700,
)
```

Voici quelques unes des question qui peuvent être posées sur cette base:

(1) "Alice est elle journaliste?"

```
: (exist X (journaliste allinst)
  (if (X nom = alice)
    T))
= T
```

(2) "Alice est elle pigiste?"

```
: (journaliste pigiste alice)
```

= T

(3) "pour quelle revue travaille alicé"

```
: (revue travaille alicé)
(MICRO-SYSTEMES MONDE-INFORMATIQUE)
```

Les questions peuvent se formuler de manière diverses. Dans l'exemple précédant les revues étaient explorées en premier. Dans l'exemple suivant on recherche d'abord sous quelle référence a été mémorisée 'Alice'.

```
: (alice ref? | revues ?)
= (MICRO-SYSTEMES MONDE-INFORMATIQUE)
```

(4) "Quelle est la périodicité des revues où travaille Alice?"

```
: (forall X (journaliste code? alicé | revues ?)
;les caracteres [ et ] indiquent qu'il y a construction d'une liste;
[X (X periodicité ?)] )
= ((MICRO-SYSTEMES MENSUEL) (MONDE-INFORMATIQUE HEBDOMADAIRE))
```

(5) "quelles sont les revues ayant au moins un journaliste age de moins de 35 ans et qui a écrit plus d'un article?"

```
: (forall X (revues allinst)
  (if (exist Y (X journalistes ?)
    (if (and (Y age < 35)
      (Y articles n? | > 2))
      X))))
= (ORDIN-INDIV MICRO-SYSTEMES MONDE-INFORMATIQUE)
```

Cet exemple a servi à montrer les mécanismes de base de MERING: la construction des attributs, l'utilisation des fonctionnelles, et la création de méthodes pour simplifier l'expression des requêtes sur la base.

Les exemples suivants vont nous permettre d'aller plus loin dans la présentation des particularités de ce langage.

I.2 Questionnaire intelligent: une réservation de places sur le chemin de fer

Les schémas s'accordent très bien à toutes les applications de ce type "questionnaire intelligent", qui mettent en scène un dialogue entre ordinateur et être humain, et qui ont pour objet de déterminer un ensemble d'informations nécessaires à l'accomplissement d'une tâche particulière: planning de rendez-vous, NUDGE (Goldstein.77), réservation de places d'avion, GUS (Bobrow.77-a), etc...

Afin de montrer que le langage MERING est capable de remplir aisément ce type de tâche, nous avons créé, à titre d'exemple, un petit système d'interrogation, spécialement destiné à la réservation de places de trains. Ce système présente beaucoup d'analogies avec le système GUS écrit en KRL par Bobrow et Winograd, particulièrement en ce qui concerne la représentation des connaissances.

Les mécanismes de contrôles de notre système ont été simplifiés: il n'y a pas d'analyse du langage naturel. De ce fait les réponses de l'utilisateur ne peuvent prendre que deux formes: soit correspondre directement à l'une des valeurs que réclame le programme, soit être une transmission MERING.

En effet, lorsque le message retourné par l'utilisateur ne convient pas, le système suppose généralement qu'il s'agit d'une transmission à évaluer (pour obtenir des informations complémentaires par exemple). Dans cette application nous formulons l'hypothèse que ces transmissions pourraient être produites par un analyseur de langage naturel qui recevrait des messages en Français.

Nous n'avons pas cherché ici à exposer un système complet de réservation de places, mais à montrer comment le langage MERING peut être utilisé pour représenter des connaissances

dans un tel contexte.

L'ensemble de la base est constituée de six types qui décrivent l'ensemble des concepts nécessaires à son fonctionnement: dialogue, voyage, trajet, date, horaire et personne. L'organisation de cette base de travail est donnée ci-après.

Un dialogue, c'est à dire le remplissage d'un questionnaire, est initialisé par la transmission:

(dialogue => | questionne)

qui crée un représentant et envoie le message questionne. Il suffit ensuite de répondre aux questions qui sont posées pour que les attributs se remplissent et que l'interrogatoire touche à son but.

La structure globale de contrôle est gouvernée par les méthodes **questionne** qui indiquent, pour chacune des entités de la base, la séquence générale des actions à accomplir lors du déroulement des questions: cette séquence est généralement une suite de requête de remplissage des attributs (applicateur ?).

Les interrogations donnent souvent lieu à des déclenchement d'entités, c'est à dire à la création de représentants désireux de voir leurs attributs recevoir des informations recueillies par l'intermédiaire de questions posées au client, ou bien déduites par inférence ou par calcul.

Cette gestion des interrogations et des inférences est réalisée directement en utilisant les mécanismes de base du langage MERING. Par exemple, examinons le cas du remplissage d'un représentant de date:

En premier lieu, la méthode **questionne** affiche un petit texte au terminal ("de quelle date s'agit il?"), puis envoie l'applicateur **question** à l'attribut jour.

Cette transmission a pour effet d'afficher le texte associé à la facette de l'attribut en question ("de quel jour s'agit il ?"), et de demander l'introduction d'une donnée de la part de l'utilisateur.

L'information, une fois reçue, tente d'être affectée à l'attribut **jour**. Or celui ci signale qu'il désire, comme donnée, un nombre compris entre 0 et le nombre maximum de jours que comporte un mois. Si la réponse est valable, la valeur est simplement affectée, et le contrôle redonné à l'entité qui a déclenché la demande d'une date.

En revanche, si la valeur ne convient pas, la facette **\$si-err-ajout** est évaluée. Celle ci suppose que cette valeur est peut être un jour de la semaine, et qu'il faut essayer de la placer dans l'attribut **joursem**.

S'il s'agit bien d'un jour de la semaine (mardi par exemple), cette valeur est affectée à l'attribut, et le réflexe **\$si-ajout** déclenché, lequel sert à mettre à jour l'attribut **jour** grâce à une fonction qui calcule la date des jours de la semaine à partir de la date du jour (fonction **datejour** écrite en LISP pour la circonstance).

Si la donnée ne peut être affectée à l'attribut **joursem**, le réflexe **\$si-err-ajout** associé à cet attribut prend en charge la gestion de cette information. Si la valeur est un atome ou la liste vide, le système demande de réitérer la question auprès du client. En revanche, cette donnée est une liste, le système suppose qu'il s'agit d'une transmission, et tente de l'évaluer. Généralement cette transmission correspond à une demande supplémentaire d'informations de la part du client.

Après évaluation de cette transmission, le système redonne la main à l'entité concernée en envoyant le message (% questionne).

Le type de mécanisme qui vient d'être décrit est très utilisé dans cette application. Souvent, un attribut est constitué de 4 facettes:

- une facette **\$dom** ou **\$req** qui contient les contraintes que doivent satisfaire les réponses de l'utilisateur.
- une facette **\$si-req** qui comprend les différentes actions qui doivent être accomplies pour remplir un attribut: impression d'un message (par évaluation de l'applicateur **question** qui utilise le message placé dans la facette **\$question**), ou création d'un représentant qui devra lui même être rempli.
- une facette **\$question** qui comporte le message à imprimer à l'utilisateur.
- une facette **\$si-err-ajout** qui indique ce qu'il faut faire si la réponse ne correspond pas. Par défaut, le système essaye de l'évaluer avant de reprendre son questionnaire là où il s'était arrêté.

Parfois une information est placée par défaut. Dans ce cas le système demande à l'utilisateur de la valider, en utilisant l'applicateur **validation**.

Les entités s'appellent les unes les autres jusqu'à l'aboutissement du questionnaire. L'entité la plus générale est 'dialogue'. Celle ci nécessite de connaître son interlocuteur, la date et le lieu du dialogue, ces deux dernières informations étant fournies par défaut. De plus elle désire remplir son attribut qui supporte le sujet du dialogue, c'est à dire la description du voyage.

Un voyage est constitué d'un seul ou de deux trajets selon que le voyageur désire un aller-simple, ou un aller-retour. Cette indication entraîne le déclenchement d'un seul ou de deux représentants du type **trajet**: aller et/ou retour.

Un trajet est caractérisé par son lieu d'origine et son lieu de destination. Dans un trajet aller, l'origine et la

destination du trajet correspondent à celles du voyage, alors que pour un trajet retour origine et destination sont inversées. Les critères sont précisés lors de la création de ces entités.

D'autre part un trajet est décrit par sa date, le type de place choisi- fenêtre, couloir ou indéfini (à ce propos, est considéré comme indéfini toute réponse qui n'est pas une transmission à évaluer)- et surtout l'horaire désiré. Celui-ci est déterminé en fonction des horaires disponibles qui sont opposés à l'aide d'une fonction **calcul-beure**, qui prend en compte le jour, le lieu d'origine et le lieu de destination du trajet.

Le dialogue est considéré comme achevé lorsque toutes les requêtes ont été menées à bien. On suppose alors que la présentation du voyage est suffisante, et que des mécanismes automatisés de réservation, comme ceux qui sont en usage à la CF, peuvent être mis en oeuvre.

Nous présentons ci-dessous un exemple de dialogue et l'ensemble des entités de la base nécessaires à cette application.

```

libération du système de recouvrement d'erreur ;
$slotini <-fac: '$sys-err-ajout
              '(meval ## & % questionne))

```

```

$objet date
  $valeur mois
  $si-req (datedujour mois ?),
  $valeur $nombre
  $si-req (% # question)
  $req (and (## > 0)(## < (ce mois jourmax ?))
  $question (" de quel jour s'agit il ?")
  $si-err-ajout (% joursem <- ##),
  $dom $nombre
  $si-req (datedujour annee ?),
  $valeur (lundi mardi jeudi vendredi samedi dimanche)
  $si-req (calculjour)
  $si-ajout (% jour <- (datejour ##)),

```

```

questionne methode: () ("precisez moi la date "
                        & % jour ?),
)

(type => $objet mois
  jourmax $default 31,
)

(mois => janvier)

(mois => fevrier
  jourmax <- 28
; On ne tient pas compte des ;
; ann{es bissextiles ;
)

(mois => mars)

(mois => avril
  jourmax <- 30
)

(mois => mai)

(mois => juin
  jourmax <-30
)

(mois => juillet)

(mois => aout)

(mois => septembre
  jourmax <- 30
)

(mois => octobre)

(mois => novembre
  jourmax <- 30
)

(mois => decembre)

```

```

type => $objet personne
  nom      $dom $chaine
           $si-req (% # question)
           $question (" quel est votre nom?"),
questionne methode: () (% nom ?),

```

```

type => $objet dialogue
  client   $dom personne
           $si-req (personne =>
                    | context <- %
                    | questionne),
  date     $dom date
           $si-req (datedujour),
  lieu     $dom ville
           $default paris,
  sujet    $dom voyage
           $si-req (voyage => | context <- % | questionne),
  debut    methode: () (print " Bonjour , je peux vous aider"
                        & print " dans le choix de votre voyage "
                        & % quest),
  questionne methode: () (forall X '(date lieu sujet client)
                        (% X ?)
                        & "au revoir"),

```

```

pe => $objet voyage
  origine  $dom ville
           $si-req (ce context lieu ?),
  destin   $dom ville
           $si-req (% # question)
           $question (" ou voulez vous aller ?"),
  typetrajet $valeur (aller-retour aller-simple)
           $si-req (% # question)
           $question
           ("voulez vous un aller-simple ou un aller-retour")
  $si-err-ajout
    (if (or (null ##)
            (atom ##))
        (print "je suppose qu'il s'agit d'un aller-simple"
              & % # validation T)
        (meval ## & % questionne))
  $si-ajout (% aller ?
            & if (## = aller-retour)
                (% retour ?)),
  aller    $dom trajet

```

```

$si-req (trajet =>
    context <- %
    lieu-origin <- (ce origine))
    lieu-destin <- (ce destin)
    questionne),
retour $dom trajet
$si-req (trajet =>
    context <- %
    lieu-origin <- (ce destin))
    lieu-destin <- (ce origin)
    questionne),
questionne methode: () (% origine ?
    & % destin ?
    & % typetrajet ? ),
)

(type => $objet horaire
    depart $dom heure,
    arrivee $dom heure,
)

(type => $objet trajet
    lieu-origin $dom ville,
    lieu-destin $dom ville,
    date $dom date
    $si-req (print "quand voulez vous aller de "
        (ce origin) "a" (ce destin)
        & date => | questionne),
typeplace $valeur (fenetre couloir indif)
$si-req (% # question)
$question (" voulez vous une place fenetre ou couloir ?")
$si-err-ajout (if (or (## nul?)
    (atom ##))
    (% typeplace <- indif)
    (meval ## & % questionne))
$si-ajout (if (## = indif)
    (print "vous etes sur d'etre indifferent ?"
    & % # validation)),
h-possible $dom horaire
$si-req (calcul-horaire (ce date jour ?)
    (ce lieu-origin)
    (ce lieu-destin)),
heure-choix $dom heure
$si-req (print "voici les heures possibles de depart :"
    & print (% heures-dep)
    & % # question)
$req (exist X (ce h-possible)
    (X depart = ##))

```

```

$question ("quelle heure choisissez vous ?")
$si-err-ajout (if (or (null ##)
                    (atom ##))
              (print "horaire impossible" & % # question)
              (meval ## & % questionne)),
questionne methode: () (% date ?
                      & % typeplace ?
                      & % heure-choix ?),

quelques methodes utilitaires pour obtenir des informations ;
plementaires (cette liste n'est pas exhaustive) :

obtenir toutes les heures de depart ;
heures-dep methode: () (forall X (ce h-possible)
                      (X depart ?)),

obtenir toutes les heures d'arrivee ;
heures-arr methode: () (forall X (ce h-possible)
                      (X arrivee ?)),

obtenir une heure d'arrivee sachant une heure de depart ;
arrivee methode: (H) (exist X (ce h-possible)
                    (if (X arrivee = H) X)),

obtenir une heure de depart sachant une heure d'arrivee ;
depart methode: (H) (exist X (ce h-possible)
                    (if (X arrivee = H) X)),

> datedujour
ls    <- decembre,
ur    <- 8,
nee   <- 1983,
ursem <- jeudi,

```

; Exemple de fonctionnement de la base de questionnaire
pour la reservation de places de chemin de fer ;

: (dialogue => | questionne)

" Bonjour , je peux vous aider"

" dans le choix de votre voyage "

"ou voulez vous aller ?"

: toulouse

"voulez vous un aller-simple ou un aller-retour"

: bof

"je suppose qu'il s'agit d'un aller-simple"

"Etes vous d'accord ?"

: non

"voulez vous un aller-simple ou un aller-retour"

: aller-retour

"quand voulez vous aller de " PARIS "a" TOULOUSE

"precisez moi la date "

"de quel jour s'agit-il"

: mercredi

; le jour est calculé à partir d'informations ;

; par défaut concernant la date du jour ;

" voulez vous une place fenetre ou couloir"

: assise

" Etes vous sur d'etre indifferent ?"

: non

" voulez vous une place fenetre ou couloir"

fenetre

" voici les heures possibles"
(10H20 14H35 17H46 21H12)

"quelle heure choisissez vous ?"

; quels sont les horaires d'arrivee ;
(trajet lastinst | heures-arr) ; une transmission a evaluer ;
(18H12 22H36 0H56 7H15)

"quelle heure choisissez vous ?"

11H23

"horaire impossible"

"quelle heure choisissez vous ?"

17H46

quand voulez vous aller de " TOULOUSE "a" PARIS

"precisez moi la date "

"de quel jour s'agit-il"

29 ; le mois et l'année sont calculées par défaut ;

"voulez vous une place fenetre ou couloir"

"Etes vous sur d'etre indifferant ?"
OUI

" voici les heures possibles"
(10H20 21H12)

"quelle heure choisissez vous ?"

; je voudrais savoir a quelle heure arrive le train de 10H20 ;
(trajet lastinst | h-arrivee 10H20) ; une transmission a evaluer;
19H04

"quelle heure choisissez vous ?"
21H12


```

"quel est votre nom?"

: Dupont

= "au revoir"

; Examinons la base resultante de ces interrogations ;

: (dialoguel print)
(DIALOGUE => DIALOGUEL
  DATE    <- (DATEDUJOUR),
  LIEU    <- (PARIS),
  SUJET   <- (VOYAGE1),
  CLIENT  <- (PERSONNEL),
)
= DIALOGUE

: (voyage1 print)
(VOYAGE => VOYAGE1
  ORIGINE    <- (PARIS),
  DESTIN     <- (TOULOUSE),
  TYPETRAJET <- (ALLER-RETOUR),
  ALLER      <- (TRAJET1),
  RETOUR     <- (TRAJET2),
)
= VOYAGE1

: (forall X (trajet allinst) (X print))
(TRAJET => TRAJET1
  LIEU-ORIGIN <- (PARIS),
  LIEU-DESTIN <- (TOULOUSE),
  DATE       <- (DATE2),
  TYPEPLACE  <- (FENETRE),
  HEURE-PROP  <- (10H20 14H35 17H46 21H12),
  HEURE-CHOIS <- (17H46),
)

(TRAJET => TRAJET2
  LIEU-ORIGIN <- (TOULOUSE),
  LIEU-DESTIN <- (PARIS),
  DATE       <- (DATE3),
  TYPEPLACE  <- (INDIF),
  HEURE-PROP  <- (10H20 21H12),
  HEURE-CHOIS <- (21H12),
)

```

TRAJET1 TRAJET2)

forall X (date allinst) (X print))

E => DATEDUJOUR

MOIS <- (SEPTEMBRE),

JOUR <- (26),

ANNEE <- (1983),

JOURSEM <- (LUNDI),

E => DATE2

JOUR <- (28),

JOURSEM <- (MERCREDI),

E => DATE3

JOUR <- (29),

DATEDUJOUR DATE2 DATE3)

mois et l'année sont par défaut ;

x de la date du jour. ;

sont placés dans les attributs ;

respondants lors d'une requête ;

te3 mois ?)

TEMBRE

te3 annee ?)

3

te3 print)

E => DATE3

JOUR <- (29),

MOIS <- (SEPTEMBRE),

ANNEE <- (1983),

33

III.3 Un jeu d'aventure

Cet exemple est destiné à montrer comment il est possible de construire un micro-monde en MERING, et à présenter une utilisation des mécanismes de réflexes. En aucun cas il ne se veut une étude générale sur les principes mis en oeuvre dans les jeux d'aventure.

L'Intelligence Artificielle aime construire des micro-mondes sur lesquels elle est à même de raisonner (certains disent même qu'elle ne sait pas faire autre chose). Depuis les mondes de "boîtes" caractéristiques des robots du style de SHRDLU (Winograd.72), ou de LAWALY (Siklossy.73) jusqu'aux mondes quasi réels des systèmes de compréhension de SCHANK (Schank.77)(Schank.81), il s'agit toujours de se définir un domaine dont les lois de fonctionnement sont bien connus.

C'est pourquoi nous avons pris comme exemple un univers de jeu d'aventure. Depuis "Donjons et Dragons" ces jeux font intervenir des personnages qui accomplissent un voyage dans les entrailles de la terre, à l'intérieur d'un réseau de souterrains constitués de salles reliées les unes aux autres .

Ces salles sont remplies d'objets divers (armes, trésors, ustensiles, outils, etc...) lesquels seront souvent amenés à jouer un rôle dans un autre épisode de cette aventure. Par exemple, trouver une clef peut permettre d'ouvrir une porte, une lanterne d'éclairer un passage sombre, etc... Il est donc essentiel que les aventuriers puissent emmener avec eux un maximum de ces bagages.

Les personnages sont caractérisés par leurs qualités intrinsèques: force, courage, vitalité, vitesse, capacité à employer certaines armes, dons spéciaux, etc... Une troupe est composée de quelques uns de ces personnages: guerrier, nain,

sorcier, elfes, etc...

Pendant ce voyage, les personnages sont parfois amenés à faire des rencontres amicales ou inamicales. Dans ce dernier cas, un combat s'engage dont l'issue dépend de la force, du courage, de la puissance et de la vitalité de chacune des parties.

Dans l'exemple qui est proposé ici, les personnages qui participent à cette aventure, comme les objets qu'ils transportent ou les éléments du décors, sont représentés à l'aide d'entités MERING. La hiérarchie des types est schématisée par la figure 3.1

Le souterrain est constitué de **salles** séparées par des **issues**. Dans certaines d'entre elles se trouvent des **objets**, qui pourront servir à franchir des issues fermées. Par exemple, la clef dorée ouvre la porte dorée, une planche permet de franchir une crevasse, une lanterne de traverser un boyau obscur. La possession de l'objet associé à la difficulté est une condition à la fois nécessaire et suffisante pour passer un obstacle. Le plan du souterrain que nous avons utilisé est donné figure 3.2.

Les personnages sont amenés au cours de leur voyage à accomplir des actions: aller dans une salle, prendre ou déposer un objet, franchir une issue, etc...

Chacune de ces notions est représentée par un comportement (une méthode) associé au type **actant** et dont les entités **personnage** et **groupe** sont des sous-types. Ainsi, l'action **va** est donnée sous la forme d'un message envoyé à l'entité en question. Par exemple:

(galark va salle1)

est un message qui demande au personnage **Galark** d'entrer dans la salle numéro 1. Evidemment cette opération ne peut être effectuée que si certaines conditions sont remplies: l'existence d'un passage entre l'endroit où se trouve le guerrier et la salle où il doit se rendre, et surtout que l'issue qui sépare ces deux lieux soit franchissable, c'est à

Figure 3.1 Hiérarchie des entités utilisées dans le jeu d'aventure

```

($objet
  (objet
    (arme
      (épée --> épée1 épée2 ...)
      (lance --> lancet lance2 ...))
    (clef --> clef-or clef-argent clef3 ...)
    (planche --> planchel ...)
    (lanterne --> lanternel ...)
    (trésor --> le-trésor))
  (actant
    (personnage
      (guerrier --> Galark)
      (Elfe --> Eldern)
      (Nain --> Trun)
      (Mage --> Nelquior)
      (Orque --> orquel orque2 ...))
    (troupe
      (troupe-orque --> tr-orq1 tr-orq2 ...)
      (compagnie --> la-partie ...))
  (issue
    (couloir --> couloir1 couloir2 ...)
    (porte --> portel ...)
    (crevasse --> crevassel ...)
    (boyau-noir --> boyaul ...))
  (salle --> extérieur sallel salle2 salle3 ...)
  (réseau --> souterrain))

```

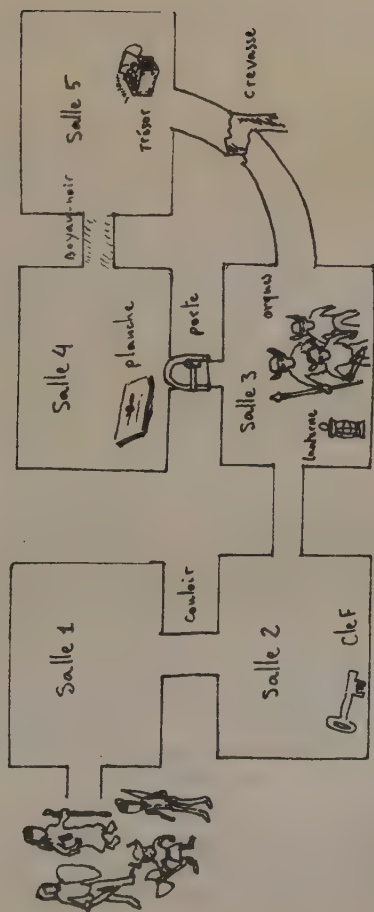


Figure 3.2. - le Souterrain

dire que l'issue soit ouverte ou que Galark détienne l'objet qui permettent de franchir l'obstacle.

Une fois l'opération réalisée, la question:

(galark lieu ?)

répond alors: sallet. Si celle ci n'avait pu se produire, le message "Impossible aller dans SALLE1" aurait été affiché au terminal.

Chaque personnage, à l'instar des jeux d'aventures, est défini à l'aide d'un certain nombre de critères: ses attributs. On distingue la force, la vitalité, la charge maximum qu'il peut porter, l'ensemble des objets qu'il détient, les armes dont il dispose pour se battre et la valeur des trésors qu'il possède.

Ces caractéristiques sont prises en compte lors de l'acquisition d'un nouvel objet, ou durant un combat avec les monstres qui peuplent le souterrain: un objet, pour être ramassé, doit avoir un poids inférieur à la charge libre (cette information est placée dans l'attribut **libre**) que peut porter le personnage en question. Autrement il sera laissé sur place, à moins qu'un ordre explicite ne demande à cet agent de déposer un autre instrument et de le prendre à sa place.

Les combats avec les monstres ont été réduits ici à leur plus simple expression. Une fonction d'évaluation, qui prend en compte les différents paramètres de force, de vitalité et de puissance armée des deux forces en présence, détermine l'issue des affrontements. Une fonction de pondération utilise une valeur aléatoire afin de laisser un peu de suspens à ces luttes.

Un type présente des caractéristiques intéressantes à mentionner: la **troupe**. Une troupe est un ensemble de personnages, considéré lui-même comme un actant. Cette notion permet de rassembler plusieurs individus sous une même dénomination, et ainsi de les commander tous ensembles.

Une troupe possède pratiquement les mêmes caractéristiques qu'un personnage ordinaire. Evidemment, ses propriétés ne sont dues qu'à la présence des personnages qui la

composent. De ce fait il a fallu prévoir des d'interactions entre troupes et individus. Ces liaisons "dyaniques" ont été placées dans des réflexes associés aux attributs concernés. Par exemple la force d'un groupe correspond à la somme des forces de chacun de ses composants; il en est de même pour la richesse, la charge maximum et la charge libre, etc...

Une troupe peut agir: n'est elle pas elle aussi un **actant**. Mais une action pour une troupe n'a pas la même signification que pour un personnage: lorsqu'une troupe va dans une salle, tous les individus qui la composent se déplacent avec elle; quand un objet est pris, il faut trouver une personne qui soit en mesure de s'en charger. Cette opération est effectuée de manière peu démocratique: le premier individu susceptible de pouvoir porter un tel poids est choisi d'office.

Appartenir à une troupe entraîne une certaine discipline, s'en éloigner conduit à la solitude. Un individu qui se sépare de sa troupe (en allant dans une autre salle) se met en péril: sera-t-il assez fort pour venir à bout d'une troupe de monstres. L'entité **troupe** prend en compte la séparation d'avec l'un des siens: sa force, vitalité et autres attributs s'en verront diminués d'autant.

Le programme fonctionne selon deux modes: dans le premier, l'utilisateur indique aux actants les actions à accomplir. Il décide de toutes les opérations à réaliser: aller dans une salle, prendre ou déposer un objet, etc... Le deuxième mode est introduit par le comportement **explorer** qui demande à l'actant en question (il peut s'agir de la troupe comme d'un personnage particulier), de découvrir un moyen pour parvenir au trésor qui se trouve dans l'une des salles.

Evidemment, des rencontres indésirables peuvent l'amener à être détruit. La démarche utilisée est celle d'un déplacement dans un labyrinthe avec "retour en arrière" (backtrack). Tous les objets rencontrés sont automatiquement saisis et ajoutés aux équipements que l'actant possède déjà.

Un exemple de fonctionnement ainsi que la base de connaissance est donnée ci-dessous:

; définition des types ;

```
(type => $objet actant
chemin      $dom salle
            <- exterieur,
lieu        $dom salle
            $default exterieur
            $si-ajout (% chemin <= ## &
                      print % " entre dans la salle " ##)
            $si-suppr (print % " sors de la salle " ##),
prend       methode: (obj) (if (ce lieu objets == obj)
                              (ce lieu objets del obj
                               & % equipt <= obj)
                              (print "y'a pas de" obj "ici")),
prendtout   methode: () (forall X (ce lieu objets ?)
                        (% prend X)),
depose      methode: (obj) (if (% equipt == obj)
                              (ce lieu objets <= obj
                               & % equipt del obj)),
va          methode: (salle) (if (% franchir? (ce lieu issue? salle))
                              (% lieu <- salle & % rencontre)),
franchir?   methode: (V) (if (V nul?)
                             (print "Impossible franchir" (V is) &
                              print % "n'a pas de " (V moyen ?) & nil)
                             (if (ou (V etat = ouvert)
                                     (V moyen == (ce equipt)))
                                 V)),
rencontre   methode: (troup) (if (ce lieu troupes ?)
                                 (print "il y a une" (ce lieu troupes ?)
                                  & print "Ils attaquent ..." &
                                   % combat (ce lieu troupes ?))),
combat      methode: (act) (if (% force < (act forces ?))
                              (send: (print % "est vaincu !" & nil))
                              (print % " a gagne")
                              (% lieu troupes del)),
regarde     methode: () (mlet ((s seq)
                               (S <- (forall X (ce lieu issue)
                                       (forall Y (X salles ?)
                                         (if (Y = %) nil
                                              Y)))))
                           (print "Il y a" (S count) "issues")
                           (print "Elles conduisent aux salles" @S)
                           (if (ce lieu objets nul?))
```

```

(print "C'est vide ")
(print "Elle contient l'objet : "
      (ce lieu objets ?)))
%),
-voies methode: (voies) (if (voies nul?) nil
      (if (% chemin == (voies 1?))
          (% expl-voies (voies rest?))
          (% va (voies 1!))
          (ifn (% explorer)
              (ifn (voies nul?)
                  (% expl-voies voies))
              T ))),
orer methode: () (if (% but)
      (send: "but atteint !!")
      (% prendtout (ce lieu objets ?))
      (if (% expl-voies
          (forall X (ce lieu issues)
              (% franchir? X)))
          T
          (% chemin del (ce lieu))
          (% va (ce chemin last))
          nil )),
methode: () (ce lieu objets == tresor),

actant personnage
pe $dom troupe
$si-req (exist X (troupe allinst)
      (if (X personnes == %)
          X )),
$si-req (ce troupe lieu ?)
$ordre-lect ($si-req $default),
emax $dom $nombre
$default 60,
t $dom objet
$req (## poids < 10),
$si-ajout (if (ce charge? > (ce chargemax))
      (% depose ##)
      (print % "prend l'objet : " ##)),
$dom arme,
sse $dom $nombre
$default 0,
$si-lu (forall X (ce equipt)
      (if (X isa? tresor)
          (X valeur ?)
          0)
      | somme ),
a? $dom $nombre

```

```

        $si-lu (forall X (% equipt)
                (X poids ?)
                | somme ),
charlibre $dom $nombre
        $si-lu (ce chargemax - (ce charge?)),
)

(type => actant troupe
  personnes $dom personnage
        $si-lu (forall X (personnage allinst)
                (if (X lieu = (ce lieu)) X)),
  force $dom $nombre
        $si-lu (forall X (ce personnes)
                (X force ?))
                | somme),
        $si-ajout (forall X (ce personnes)
                (X force <- (## / (% personnes n?)))),
  richesse $si-lu (forall X (ce personnes)
                (X richesse ?)
                | somme),
  chargemax $si-lu (forall X (ce personnes)
                (X chargemax ?)
                | somme),
  charge? $dom $nombre
        $si-lu (forall X (ce personnes)
                (X charge? ?)
                | somme),
  equipt $dom objet
        $si-ajout (ifn (exist X (ce personnes)
                (if (X charlibre > (## poids ?)
                    (X equipt <= ## & % equipt del ##)))
                (% depose ##)
                T ),
)

(type => $objet objet
  poids $dom $nombre
        $mode $uni,
  forme $dom forme,
  couleur $dom couleur,
  partic $dom $atom,
)

(type => objet lanterne
  poids <- 3
)

(type => objet clef

```

```

poids <- 0.5

pe => objet planche
poids <- 20

pe => objet tresor
valeur $dom $nombre

pe => objet arme
poids $default 5

pe => $objet salle
issues $dom issue
  $si-req (forall X (issue allinst)
    (if (X salles == %)
      X )),
objets $dom objet,
troupees $dom troupee,
issue? (V) (mlet ((iss cell))
  (iss <- (exist X (ce issues)
    (if (X salles == V )
      X ))
  (ifn (iss nul?) T
    (print "il n'y a pas d'issue entre" % "et" V
      & nil)

pe => $objet issue
salles $dom salle,
etat $valeur (ouvert ferme),
moyen $dom $objet,

pe => issue couloir
etat <- ouvert,

pe => issue porte
etat <- ferme,
moyen <- clef,

pe => issue crevasse

```

```

    etat <- ferme,
    moyen <- planche,
)

(type => issue boyau-noir
  etat <- ferme,
  moyen <- lanterne,
)

(type => issue entree
  etat <- ouvert,
)

(type => $objet reseau
  salles $dom salle,
)

(type => troupe compagnie)

(type => troupe monstres)

; définition des représentants
1) le réseau ;

(salle => salle1)

(salle => salle2 | objets <- (clef =>))

(salle => salle3 | objets <- (lanterne =>
  troupes <- (troupe-orques =>
    nombre <- 5
    force <- 120
  )
)

(salle => salle4 | objets <- (planche =>))

(salle => salle5 | objets <- (tresor => | poids <- 40 | valeur <- 2000))

(entree => | salles <- '(exterieur salle1))

(couloir => | salles <- '(salle1 salle2))

(couloir => | salles <- '(salle2 salle3))

(boyau-noir => | salles <- '(salle4 salle5))

```

```

sse => | salles <- '(salle3 salle5))
=> | salles <- '(salle3 salle4))
=> reseaul | salles <- '(salle1 salle2 salle3 salle4 salle5))

```

les participants ;

```

gnie => partie
personnes <- (guerrier => galark | force <- 60)
personnes <= (elfe => eledern | force <- 50)
personnes <= (nain => trun | force <- 50)
personnes <= (mage => nelkior | force <- 10)

```

ici un exemple de fonctionnement de ce jeu d'aventure:
 ecution en mode manuel

```

(partie va salle1)
PARTIE entre dans la salle SALLE1
PARTIE

(partie personne ?)
(GALARK ELEDERN TRUN NELQUIOR)

(partie regarde)
Il y a 2 issues.
Elles conduisent aux salles (EXTERIEUR SALLE2)
La salle est vide
PARTIE

(partie va salle2)
PARTIE entre dans la salle SALLE2
PARTIE

(partie regarde)
Il y a 2 issues
Elles conduisent aux salles (SALLE1 SALLE3)
Elle contient l'objet : CLEF
PARTIE

```

: (partie prend clef)
GALARK prend l'objet : CLEF
; une troupe ne peut prendre un objet ;
; il faut qu'une personne s'en charge ;
= PARTIE

: (trun regarde)
Il y a 2 issues
Elles conduisent aux salles (SALLE1 SALLE3)
C'est vide

= TRUN

: (partie va salle3)
PARTIE entre dans la salle SALLE3
Il y a une TROUPE-ORQUES
Ils attaquent

PARTIE a gagne !!

: (galark regarde)
Il y a 3 issues
Elles conduisent aux salles (SALLE2 SALLE4 SALLE5)
Elle contient l'objet : LANTERNE
= GALARK

: (galark va salle5)
Impossible franchir CREVASSE
GALARK ne possede pas de PLANCHE
= GALARK

: (galark va salle1)
Il n'y a pas d'issue entre SALLE3 et SALLE1
= GALARK

: (eledern va salle4)
Impossible franchir PORTE
ELEDERN ne possede pas de CLEF
= ELEDERN
; en effet c'est GALARK qui la ;
; detient ;

: (galark depose clef & eledern prend clef)
ELEDERN prend l'objet : CLEF
= ELEDERN

: (eledern va salle4)

ELEDERN entre dans la salle SALLE4
= ELEDERN

: (partie lieu ?)
= SALLE3

: (eledern lieu ?)
= SALLE4

: (partie personnes ?)
= (GALARK TRUN NELQUIOR)
; Eledern s'est séparé des autres membres ;
; de la troupe ;

: (eledern regarde)
Il y a 2 issues
Elles conduisent aux salles (SALLE3 SALLE5)
Elle contient l'objet PLANCHE
= ELEDERN

: (partie regarde)
Il y a 3 issues
Elles conduisent aux salles (SALLE3 SALLE4 SALLE5)
ELLE CONTIENT L'OBJET LANTERNE
= PARTIE

: (eledern va salle5)
Impossible franchir BOYAU-NOIR
ELEDERN ne possède pas de LANTERNE
= ELEDERN

: (eledern prend planche)
ELEDERN prend l'objet : PLANCHE
= ELEDERN

: (eledern va (partie lieu ?))
ELEDERN entre dans la salle SALLE3
= ELEDERN

: (partie va salle5)
PARTIE entre dans la salle SALLE5
= PARTIE

: (partie but)
= T
; ils ont enfin atteint au but ;
; c'est la salle du tresor ;

Un autre exemple d'exécution de fonctionnement en mode autonome:
Afin de montrer le mécanisme de backtrack, on supprime la
lanterne qui se trouve dans la salle3

: (salle3 objets del)
= SALLE3

: (partie lieu ?)
= EXTERIEUR

: (partie explorer)
PARTIE entre dans la salle SALLE1

PARTIE entre dans la salle SALLE2

GALARK prend l'objet : CLEF

PARTIE entre dans la salle SALLE3

PARTIE entre dans la salle SALLE3
Il y a une TROUPE-ORQUES
Ils attaquent

PARTIE a gagne !!

PARTIE entre dans la salle SALLE4

GALARK prend l'objet : PLANCHE

Impossible franchir BOYAU-NOIR
PARTIE n'a pas de LANTERNE

PARTIE sors de la salle SALLE4

PARTIE entre dans la salle SALLE3

PARTIE entre dans la salle SALLE5

= "But atteint !!"

Enfin, un essai avec GALARK tout seul:

(galark explorer)

GALARK entre dans la salle SALLE1

GALARK entre dans la salle SALLE2

GALARK prend l'objet : CLEF

GALARK ente dans la salle SALLE3

Il y a une TROUPE-ORQUES

Ils attaquent !!

GALARK est vaincu

; il ne fait pas le poids devant ;

; une troupe d'Orques ;

= NIL ; et tout s'arrete ;

III.4 Programmation par contrainte

La programmation classique a été constituée autour de la notion de processus: les programmes traitent des données fournies en entrée afin de produire un résultat ou bien d'effectuer une tâche déterminée.

D'un autre côté, un grand nombre de systèmes concrets et abstraits sont donnés sous la forme d'un ensemble de contraintes appliquées sur certaines variables. Ce que l'on nomme, dans le domaine physique et mathématique, sous le nom d'équation.

Par exemple, l'équation $X + Y = 5$, ne stipule pas l'ordre de calcul. Il s'agit seulement d'une connaissance déclarative sur les calculs qui peuvent être effectués. En particulier cela signifie, que sachant X on en déduit Y , ou bien que la connaissance de Y infère X .

Ce type de description est particulièrement intéressant pour décrire des structures associées à des ensembles de contraintes: en particulier des systèmes concrets qui font intervenir des variables numériques, c'est à dire dont la modélisation porte aussi bien sur des flux quantitatifs que sur leur structure (physiques, électriques, économiques).

Des études dans ce domaine ont été accomplies en particulier par STEELE (Steele.78)(Steele.80) donnant lieu à l'établissement de systèmes destinés à la construction et à la résolution de circuits électroniques.

Nous allons voir comment en MERING, il est possible d'utiliser des informations portant sur certains types de contraintes, et comment il est possible d'organiser un réseau d'entités reliées par contraintes, sans sortir du modèle de base que nous avons développé au chapitre II. En particulier nous verrons l'importance de la notion d'applicateur et de son

interaction avec celle des réflexes.

A cet effet, nous montrerons comment ces mécanismes peuvent être efficacement employés pour la modélisation de les circuits électroniques.

Par rapport au système développé par STEELE, notre système présente les avantages suivants:

- Simplification de la description des composants et des circuits
- Compilation des contraintes d'égalité sous la forme de réflexes
- Intégration totale des modèles au langage: chaque circuit est représenté par une entité MERING, les contraintes sont des applicateurs, les liaisons dynamiques sont gérées par réflexes.
- Cette caractéristique conduit à une simplification du système: en effet, l'écriture d'une telle application est immédiate à partir de MERING, alors que STEELE a dû écrire un programme particulier qui sert uniquement à modéliser de tels systèmes dynamiques.

III.4.1 La constitution des composants de base

Un circuit électronique est composé d'éléments passifs et actifs qui interagissent entre eux, et qui définissent donc des contraintes sur l'ensemble des valeurs permises: placer une donnée à l'une des terminaisons du circuit conduit à la propagation d'informations le long des liaisons, en tenant compte de la structure des éléments.

Par exemple un additionneur, ou un multiplieur, contraint les valeurs dans un certain rapport. la figure 3.3 montre le schéma de ces circuits. Ceux ci sont composés de trois terminaisons, nommées A1, A2 et RES.

En premier lieu, nous créons un représentant du type 'additionneur'.

```
: (additionneur => som1)
= SOM1
```

maintenant nous essayons de connaître les valeurs qu'il contient:

```
: (som1 res ?)
= 0
```

Puisque nous n'avons pas précisé de valeur particulière, le résultat est nul. Nous allons maintenant affecter une valeur que doit prendre l'une des terminaisons, à savoir la terminaison RES.

```
: (som1 a1 <- 5 | res ?)
= 5
; la borne A1 vaut maintenant 5 ;
```

```
: (som1 a2 ?)
= 0
; A2 est outjours nulle ;
```

```
: (som1 res <- 8 | a2 ?)
= 8
; mais en affectant la valeur 8 a RES, ;
; A2 vaut maintenant 3 ;
```

```
: (som1 print)
(ADDITIONNEUR => SOM1
  A1      <- 5,
  A2      <- 3,
  RES     <- 8,
```

```
)
= SOM1
```

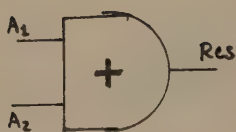


Figure 3.3 : Un additionneur et un multiplieur

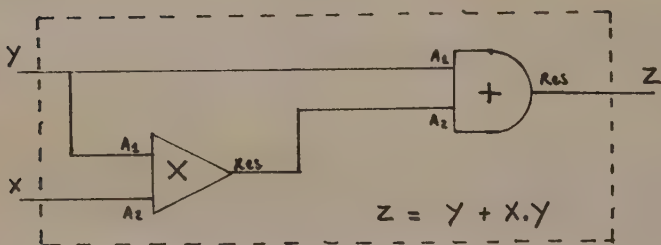


Figure 3.4 : schéma d'un circuit électrique

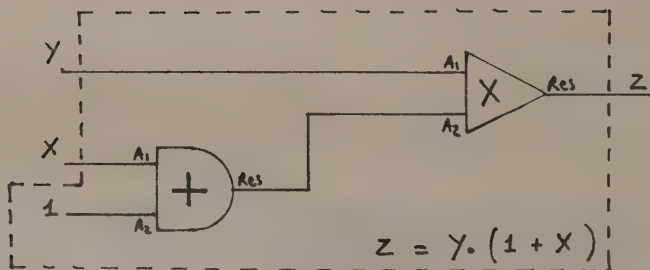


Figure 3.5 : schéma du circuit équivalent

```
: (soml a2 <- 7 | a1 ?)
= 1
; changer la valeur de l'une des bornes, ici A2, ;
; modifie la valeur de toutes les autres ;

: (soml print)
(ADDITIONNEUR => SOM1
  A1    <- 1,
  A2    <- 7,
  RES   <- 8,
)

= SOM1
```

La construction d'un type additionneur est aisément réalisée par l'intermédiaire de réflexes attachés aux terminaisons et qui prennent en compte leurs relations structurales. Le réflexe employé est contenu sur la facette \$si-lu et se trouve déclenché avant toute lecture de l'attribut.

Afin d'éviter la circularité dans les appels de réflexes, la lecture des valeurs est réalisée de manière interne grâce à l'applicateur ?? qui court circuite le déclenchement de ce réflexe. Le type 'additionneur' s'exprime donc ainsi:

```
(type => $objet additionneur
  termin <- (al a2 res),
  al      $dom : $nombre
          $default 0
          $si-lu ((% res ??) - (% a2 ??)), ; al = res - a2;
  a2      $dom : $nombre
          $default 0
          $si-lu ((% res ??) - (% al ??)), ; a2 = res - al;
  res     $dom : $nombre
          $default 0
          $si-lu ((% al ??) + (% a2 ??)), ; res = al + a2;
)
```

Un multiplieur s'exprime de manière semblable. Nous devons simplement faire attention à la valeur du diviseur. Si elle est nulle, le résultat est indéterminé. En fait pour nous, sa valeur

sera attribuée arbitrairement comme très grande, c'est à dire que nous supposons qu'une division par zéro peut se ramener à une division par un infiniment petit. Cette approximation est valable dans le cas de systèmes physiques.

```
(type => $objet multiplieur
  termin <- (al a2 res),
  al      $dom  $nombre
          $default 1
          $si-lu (if (% a2 = 0) HUGE
                    ((% res ??) / (% a2 ??)), ; al = res / a2;
  a2      $dom  $nombre
          $default 1
          $si-lu (if (% al = 0) TINY
                    ((% res ??) / (% al ??)), ; a2 = res / al;
  res     $dom  $nombre
          $default 1
          $si-lu ((% al ??) * (% a2 ??)),      ; res = al * a2;
)
```

III.4.2 Composition des circuits

Un élément unique ne présente pas d'intérêt par lui-même. Il n'acquiert son sens qu'en liaison avec d'autres composants afin de constituer des systèmes plus complets. Par exemple, nous pourrions connecter notre sommateur à un multiplieur 'mult1' comme le montre la figure 3.4, et qui correspond à l'équation:

$$Z = Y + (X * Y)$$

Pour relier des terminaisons appartenant à des composants différents, nous avons créé un applicateur particulier : $\langle \Rightarrow \rangle$ dont le rôle est de propager les valeurs disponibles aux terminaisons, comme le feraient des fils électriques. Il sert donc à décrire un point de contact entre deux bornes de composants. D'autre part nous devons pouvoir écrire que certaines terminaisons sont maintenant internes au circuit et que leur valeur doit être calculée automatiquement. Pour cela

utilisons l'applicateur **interne** qui indique à l'élément
calculer cette valeur chaque fois qu'un nombre est déposé sur
des autres terminaisons. De ce fait le circuit est défini
:

```
(multiplieur => mult1)
= MULT1

(mult1 res <=> som1 a1 & mult1 a2 <=> som1 a2)
= MULT1
; la borne A1 du sommateur est reliée à la borne RES;
; du multiplieur, et les deux bornes A2 des deux ;
; composants sont en contact. ;

(mult1 res interne & som1 a1 interne)
= SOM1
; RES de MULT1 et A1 de SOM1 sont d{crits comme internes;
; au circuit, et deviennent donc inaccessibles. ;
```

aut ensuite initialiser les valeurs des terminaisons du
it:

```
(mult1 a1 <- 3 | a2 <- 2)
= MULT1
; initialisation ;

(som1 res ?)
8

(som1 res <- 10 & mult1 a1 ?)
4
la modification des valeurs se propage;

(som1 a2 <- 4 & mult1 a1 ?)
1.5
```

ons ce que sont devenus les composants 'som1' et 'mult1':

```

: (som1 print & mult1 print)      ,
(ADDITIONNEUR => SOM1
  A1      <-      6
          $SI-AJOUT (MULT1 RES <- ##),
  A2      <-      4
          $SI-AJOUT (MULT1 A2 <- ## & SOM1 A1 ?),
  RES     <-     10
          $SI-AJOUT (SOM1 A1 ?),
)

(MULTIPLIEUR      => MULT1
  A1      <-     1.5
          $SI-AJOUT (MULT1 RES ?),
  A2      <-      4
          $SI-AJOUT (SOM1 A2 <- ## & MULT1 RES ?),
  RES     <-      6
          $SI-AJOUT (SOM1 A1 <- ##),
)

= MULT1

```

L'affichage des entités 'som1' et 'mult1' laisse voir non seulement les valeurs des terminaisons des composants, mais aussi les "liens dynamiques", créés par les réflexes, constituant ainsi de véritables dépendances qui s'exercent entre les éléments du circuit.

Ces réflexes ont été placés lors de la déclaration des caractéristiques du réseau électrique. Ils sont donc le résultat de l'évaluation des applicateurs <=> et interne. Il s'agit d'une véritable compilation de réflexes, c'est à dire d'une transformation de connaissances déclaratives en connaissances procédurales.

Ces applicateurs qui servent à la compilation de réflexes pourraient se décrire ainsi:

```

<=> applic: (O S) (if (% # nulfac? '$si-ajout)
                  (% # <=fac | $si-ajout (O S <- ##))
                  (% # <=fac | $si-ajout (& O S <- ##))
                & if (O S nulfac? '$si-ajout)
                  (O S <=fac | $si-ajout (% # <- ##))
                  (O S <=fac | $si-ajout (& % # <- ##)))

```

```
interne applic: () ($seq => $seqterm | <- (% termin ?)
& $seqterm del #
& forall -X- ($seqterm ?)
  (if (% -X- nulfac? '$si-ajout)
    (% -X- <=fac| $si-ajout (% # ?))
    (% -X- <=fac| $si-ajout (& % # ?))))
```

Grâce à ces réflexes, les valeurs du circuit se propagent automatiquement d'un bout à l'autre du réseau, permettant de créer une structure complète, certaines terminaisons étant reliées à l'aide de "liens dynamiques". La déclaration des assemblages de contraintes conduit à la compilation de ces liens, et permet un calcul rigoureux à partir d'équations.

III.4.3 La définition de types de circuits

À l'aide des applicateurs $\langle = \rangle$ et *interne*, il est possible de construire des réseaux d'additionneurs et de multiplieurs de complexités quelconque. Lorsque plusieurs assemblages ont une structure identique il est intéressant de pouvoir nommer ces types d'assemblages, et de les considérer comme des composants à part entière.

Cette opération de construction et de définition de circuits à partir de sous-circuits est classique en électronique: un appareil électronique est constitué de dizaines de circuits intégrés eux mêmes composés de centaines de transistors. Les composants de base deviennent ainsi les éléments de systèmes de plus en plus complexes.

Par exemple, imaginons que le circuit que nous venons de voir se retrouve souvent dans un circuit plus complexe. Nous allons alors considérer qu'il existe un type 'mult-som' qui définit la structure de l'assemblage que nous venons de voir, et dont les représentants sont autant de circuits semblables.

Un tel circuit est composé d'une liste de terminaisons qui lui sont propres, d'un ensemble de composants et de la définition de la structure qui les unit. La description de cette

structure est placée dans la méthode **new-rep** qui est déclenchée lors de la création d'un représentant.

```
(type => circuit mult-som
; la liste des terminaisons propres au circuit ;
  termin <- (X Y Z),
  X      $dom $nombre
        $default 0,
  Y      $dom $nombre
        $default 0,
  Z      $dom $nombre
        $default 0,

; l'ensemble des composants du circuit ;
  compos <- (add mul),
  e11     $dom multiplieur,
  e12     $dom additionneur,

; la description de la structure ;
  new methode: ()
    (% e11 <- (multiplieur => )
    & % e12 <- (additionneur => )
; les liens dynamiques ;
    & (% e11 ?) a2 <=> (% e12 ?) a2
    & (% e11 ?) res <=> (% e12 ?) a1
    & (% e11 ?) res interne
    & (% e12 ?) a1 interne

; les liens entre terminaisons des composants ;
; et terminaisons du circuit ;
    & % X <=> (% e11 ?) a1
    & % Y <=> (% e11 ?) a2
    & % Z <=> (% e12 ?) res ),
)
```

Ensuite, ce circuit peut être employé comme s'il s'agissait d'un simple composant de base.

```
: (mult-som => msl)
= MS1
; creation d'un circuit de type mult-som;

: (msl X <- 3 | Y <- 2 | Z ?)
= 8
;initialisation des valeurs ;
```

```
: (msl Z <- 10 | X ?)
```

```
= 4
```

```
; les modifications se propagent bien ;
```

réalité, comme STEELE l'a montré, le circuit que nous avons
rit présente des difficultés lorsque la valeur de Y est lue,
ès qu'une autre valeur, X ou Z a été modifiée. En effet, la
pagation simple des valeurs n'est pas suffisante s'il existe
boucles possibles de calcul. Par exemple, dans notre réseau
valeur de Y peut être calculée de deux manières
férentes:

$$(1) Y = Z - XY$$

$$(2) Y = (Z - Y) / X$$

Nous constatons que ces formes font intervenir Y dans le
bre de gauche comme dans celui de droite. Dans ce cas, la
leur résultante de Y est différente selon que l'on calcule Y à
aide de la première équation ou de la seconde, et dans les
x cas celle ci est mauvaise. En effet, il ne s'agit pas d'une
ation au sens mathématique du terme, mais plutôt d'une
ectation: à gauche Y est une variable, alors qu'à droite, Y
présente le contenu d'une variable. Par exemple, pour des
eurs initiales X=3 Y=6 Z=12 on obtient:

$$(1) Y = -6$$

$$(2) Y = 2$$

rs que le résultat souhaité est 3.

Afin de pallier cette difficulté, il est nécessaire de
nsformer le circuit en un circuit équivalent; mais qui ne
sente pas de boucle de calcul. Une forme équivalente et
cte de ce réseau est donnée par l'équation:

$$Z = Y (1 + X)$$

Dans ce cas, il n'y a plus de danger. Le schéma de ce

nouveau circuit est présenté figure 3.5, et sa définition en MERING est la suivante:

```
(type => circuit mult-som2
; les terminaisons ;
  termin <- (X Y Z),
  X      $dom $nombre
        $default 0,
  Y      $dom $nombre
        $default 0,
  Z      $dom $nombre
        $default 0,
; les composants ;
  compos <- (add mul),
  e11    $dom multiplieur,
  e12    $dom additionneur,
; la structure ;
  new methode: ()
    (% e11 <- (multiplieur =>)
    & % e12 <- (additionneur =>)
    & (% e11 ?) a1 <=> (% e12 ?) res
    & (% e12 ?) a1 <- 1
    & (% e12 ?) res interne
    & (% e12 ?) a1 interne

    & % X <=> (% e12 ?) a1
    & % Y <=> (% e11 ?) a2
    & % Z <=> (% e11 ?) res ),
)
```

Maintenant il n'y a plus de problèmes:

```
: (mult-som2 => ms2)
= MS2

: (ms2 X <- 3 | Y <- 2 | Z ?)
= 8

: (ms2 Z <- 12 | Y ?)
= 3 ; l'effet de boucle a été supprimé ;
```

Afin d'éliminer les boucles du circuit, il est possible de transformer le réseau grâce à des procédés de manipulation algébrique, en s'assurant que chaque borne du circuit n'est

connectée qu'à un seul composant.

Notre but n'est pas de nous étendre sur les problèmes de réseaux (pour plus de précision à ce sujet, se référer aux travaux de STEELE qui présente une étude complète des difficultés que soulèvent de telles modélisations), mais de montrer comment, avec l'aide de réflexes et d'applicateurs (dont le but est de compiler des réflexes et de réaliser ainsi des "liens dynamiques" entre attributs), il est possible de construire des structures dont le fonctionnement est régi par un ensemble de contraintes.

Types et classes

Souvent confondues en Intelligence Artificielle, les notions de type générique et de classe logique (ou ensemble mathématique) ne peuvent cependant être substituées l'une à l'autre. Elles procèdent historiquement et logiquement de deux notions différentes.

1.1 Généralités

La notion de type abstrait prend sa source dans ce que les philosophes appellent les "universaux", qui détiennent une certaine célébrité à cause de la querelle qui a opposé pendant des siècles les tenants du réalisme, pour lesquels les universaux ont une existence indépendante à la fois des choses individuelles qui les constituent et de l'esprit de celui qui les pense, et des nominalistes qui considèrent que ces universaux ne sont que des mots, substituables à des références aux objets réels, lors de la production d'énoncés. Enfin les tenants du conceptualisme forment un groupe à part, en affirmant qu'il existe une réalité mentale, les concepts, distincte des mots, mais sans nécessaire contrepartie hors du domaine de l'esprit.

Ces trois visions ne sont pas nécessairement dépassées. Le réalisme a débouché sur le positivisme réaliste et les sciences expérimentales, notamment la physique antérieure à la révolution relativiste, le nominalisme sur la logique formelle et le formalisme logique, enfin le conceptualisme correspond à la psychologie de "schème" et d'opération mentale chère à Piaget.

Nous avons vu dans le chapitre I comment le conceptualisme se rapproche du formalisme "objet" que nous adoptons dans notre travail. Selon ce point de vue, un type est considéré comme une

entité stéréotype (autrement dit un "archétype"), qui décrit l'ensemble des caractères communs dont disposent généralement ses représentants.

Ces types correspondent à la notion commune de "concept": le résultat d'une construction mentale à partir d'un substrat réel, la synthèse d'une réalité et d'une opération d'abstraction. En effet comme le signale R.THOM (Thom.80), un phénomène pour être nommé, un "quelque chose" pour être identifié comme étant bien une chose, doit nécessairement faire l'objet d'une certaine stabilité structurelle: il est impossible de conceptualiser, et encore moins de nommer le chaos, si ce n'est en lui assignant ce terme qui recouvre tous les phénomènes indescriptibles et "indicibles".

Cependant, comme l'a montré J.PIAGET (Piaget.76), tout concept, toute idée, tout processus mental est le fruit d'une élaboration active du sujet sur son environnement. Par exemple, la notion d'objet, d'état, d'action, etc.. sont pour nous des types principaux (et non des connaissances a priori au sens Kantien) qui moulent l'ensemble de nos connaissances: tout type ultérieur sera alors classé comme dérivant d'un objet d'une classe, d'une action, etc...

L'opération qui fait dépendre un type d'un autre s'intitule **différenciation**. Cette particularisation d'un type en sous-types, cette faculté de décrire des prototypes plus spécifiques à partir d'entités plus générales est fondamentale et a été reconnue depuis Aristote sous la dénomination de hiérarchie des universaux.

En Intelligence Artificielle, il a fallu attendre les travaux de QUILLIAN (Quillian.68) sur la mémoire sémantique puis ceux de WOODS (Woods.75) et de FAHLMAN (1979) sur les réseaux sémantiques pour que de telles classifications aient enfin droit de cité. Bien que critiquées pour leur caractère arbitraire et leur non-universalité, elles ont néanmoins montré leur efficacité dans des domaines conceptuels restreints et bien structurés dont la taxinomie ne soulève pas trop de difficultés: zoologie, botanique, ameublement, nourriture, outillage, objets usuels, produits industriels, architecture, etc...

Si le champ des objets semble assez facilement constituable

de cette manière, celui des actions, et encore plus des situations, offre une certaine résistance à un découpage systématique. Néanmoins, en se restreignant à un cadre conceptuel plus étroit, il devient possible de définir une bonne, sinon unique, classification hiérarchique.

L'introduction d'héritages multiples et des classifications en treillis réduisent encore l'importance de ces difficultés en permettant notamment de résoudre la question de taxinomies multiples: 'théâtre', par exemple, peut être envisagé comme une 'sorte de spectacle' mais aussi comme une 'sorte d'établissement'. L'idée d'héritage, est caractéristique de la notion de type, et s'oppose en cela à celle de classe.

Grâce à l'héritage, tous les représentants d'un type possèdent les mêmes attributs généraux, sauf si des propriétés particulières sont précisées. Par exemple Fido, qui est un chien borgne, dispose de toutes les qualités communes aux chiens, si ce n'est qu'il ne possède qu'un oeil. La démarche de raisonnement est déductive par essence: Fido est un chien, or les chiens ont quatre pattes, donc Fido a quatre pattes.

La notion de classe est apparue plus tardivement que celle de type. Les développements de la logique mathématique associées aux réflexions des positivistes logiques, ont conduit à dégager l'idée de classe des éléments qui la comprennent. Une classe est introduite par l'opérateur ' λ ' (appelé opérateur d'abstraction de classe) qui crée une classe à partir d'objets individuels, et surtout permet ensuite de manipuler ces classes.

Pour le sens commun, la classe, ou ensemble, correspond à l'idée de réunion d'objets, de collection, à la capacité de regrouper des objets qui ont des caractéristiques communes sous une même appellation, au sein d'un même lot: l'ensemble des médecins, des étudiants de Jussieu, des personnes qui ont les yeux bleus ou des ballons ronds, en sont des exemples.

L'approche est ici inductive. Une classe est créée à partir de n'importe quelle collection d'objets. Il s'agit donc d'une notion plus globale, mais aussi plus difficile à cerner, lorsqu'on la ramène à une sémantique "naturelle": si la classe 'chien' correspond assez étroitement au type 'chien', il n'en

est plus de même des 'chiens qui passent dans ma rue, le matin'. L'ensemble des chiens qui passent le matin dans ma rue, définissent-ils un prototype en relation avec celui des chiens, et surtout méritent-ils la création d'un type?

Dans ce cas, la différence principale provient du caractère essentiellement permanent du type et de l'aspect plus abstrait de la classe. Un type n'est toujours qu'un prototype, un modèle "idéal" de ses représentants, et de ce fait trouve une correspondance dans la réalité. A l'inverse une classe n'a pas forcément de corrélat dans la réalité, car il s'agit le plus souvent d'une abstraction commode.

D'autre part, ce qui distingue les types des classes sont les opérations qui peuvent être appliquées sur ces dernières: réunion, intersection, différence symétrique, etc...

De plus, il n'existe pas à proprement parler d'héritage sur les classes, mais une notion d'inclusion. Par exemple, si la classe 'chien' a deux yeux, alors Fido possède **nécessairement** deux yeux: il n'y a pas possibilité de redéfinir une particularité, une exception, qui distingue un individu (ou une sous-classe) de la classe qui le contient (ou l'inclus).

En effet la notion d'héritage se rapproche de celle de valeur par défaut qui soulève de nombreuses questions logiques (Reiter.80)(Reiter.83), et s'avère bien trop floue pour un calcul formel.

Dans son système de représentation, NETL, FAHLMAN avait conçu la notion de "classe définie" qui se distingue de celle de prototype par le fait que la première définit un ensemble alors que la seconde décrit les membres qu'il comporte.

Il distingue les informations qui ont la propriété de définir un concept, c'est à dire qui sont des critères pour déterminer si un individu appartient bien à la classe ou non, de celles qui ne sont attribuées qu'a posteriori, déduites des autres et donc considérées comme "accessoires".

Par exemple, dans l'assertion "les médecins Britanniques et conventionnés", le fait d'être une personne ayant la profession 'médecin' et la nationalité 'Britannique' sont des propriétés de définition, tandis que le statut 'conventionné' est qu'accessoire car il peut être déduit des précédentes.

A partir de ces idées, et afin d'être à même de traiter un genre de classes, nous avons créé le type **classe** dont la définition permet de prendre en compte des descriptions en extension comme en compréhension (avec quelques restrictions sur la dernière), et surtout d'être à même de leur appliquer des opérations ensemblistes.

2 Les Classes en MERING

Par rapport à NETL, notre système présente les avantages suivants:

1. **Testage de reconnaissance** (c'est à dire le mécanisme par lequel on détermine si un individu est membre ou non d'une classe) simplifié et surtout amélioré, en l'incorporant dans notre modèle général d'entités MERING.

2. **Capacités d'effectuer des opérations ensemblistes:** intersection, union, sous-classe, etc... Toutes ces opérations donnent lieu à la création de nouvelles classes.

3. **Capacité de génération de nouveaux individus appartenant à une classe**, de manière semblable (dans la forme) à celle de la création d'un représentant de type: cette particularité est bien pratique pour constituer de nouveaux éléments: vouloir créer un médecin Britannique sous le nom de Peter revient à envoyer le message:

```
(cl-med-britan => peter)
```

De plus, à chaque type, nous associons une classe implicite (dont le nom est justement celui du type lui-même) qui est celle de l'ensemble de ses représentants. En d'autres termes, les opérations ensemblistes peuvent être réalisées sur des types (ou entre un type et une classe). Néanmoins, les entités résultantes sont nécessairement des classes. Les classes sont donc des ensembles absorbants pour ces opérations ensemblistes.

Chaque classe est un représentant du type 'classe', qui se définit en indiquant la liste des attributs supportant les informations de définitions, la liste des connaissances accessoires, puis la définition de ces attributs eux mêmes. Par exemple, la définition de la classe 'cl-med-britan' est donnée ainsi:

```
(classe => cl-med-britan
; l'attribut qui contient la liste des attributs de definition ;
  defattr      <- (profession nation),

; l'attribut qui contient la liste des attributs accessoires ;
  defacc       <- statut,

; le type en rapport avec la classe ;
  type        <- personne,

; les autres attributs ;
  profession  <- medecin,
  nation      <- grande-bretagne,
  statut      <- conventionne,
)
```

En MERING, le type **classe** est sous-type de **seq**. De lui il hérite donc des manipulations de base: ajout d'un élément, suppression, lecture, etc... Par exemple:

```
; creation d'une classe avec deux elements ;
: (classe => cl1 | <- (toto riri))
= CL1

: (cl1 ?)
= (TOTO RIRI)
```

```
; rajouter un individu ;
: (c11 <= fifi | ?)
= (TOTO RIRI FIFI)

; supprimer un element
: (toto del riri | ?)
= (TOTO FIFI)

; compter le nombre d'element ;
: (c11 n?)
= 2
```

Décrire une classe, c'est bien, la constituer c'est mieux. Par constituer nous entendons la mise en oeuvre des mécanismes qui placent les individus au sein de la classe. En premier lieu nous voulons que toutes les entités qui possèdent déjà ces caractéristiques soient placées dans l'extension de la classe, et d'autre part que tout nouvel individu détenant ces propriétés y soit ajouté.

Pour ce qui concerne le premier de ces souhaits, pas de problèmes. Une méthode y suffit. Elle a pour fonction de ramener tous les individus qui correspondent à la description.

En revanche, le second est plus délicat. FARLMAN utilise un principe simple: dès qu'un représentant est créé, il regarde si une classe n'est pas reliée au type dont il est issu. Si oui, il essaie de déterminer si le représentant en question appartient bien à une classe ou non, en le "filtrant" (c'est à dire en effectuant ce qu'il appelle un "node-matching") sur la description de la classe.

Cette solution est facile à mettre en oeuvre en MERING. Il suffit de redéfinir la méthode **new-rep** (qui, rappelons le, est déclenchée lors de la création d'un représentant) de manière à ce qu'elle réalise cette tâche. Elle s'exprime ainsi:

```
new-rep methode: () (forall X (classe allinst)
                    (X exec %))
```

la méthode **exec** servant à déterminer si un élément

appartient à la classe en question.

Cette solution est simple, mais pas très efficace: elle nous empêche d'employer la méthode **new-rep** à une autre tâche; mais ceci n'est pas trop gênant car nous pourrions trouver aisément des moyens pour passer outre cet inconvénient. Non ce qui est plus grave, c'est que ce test est alors effectué lors de la création d'un représentant quelconque, et de plus il impose de regarder toutes les classes.

Afin de pallier ces difficultés, nous allons créer des liens dynamiques (c'est à dire utiliser les réflexes) des attributs des types concernés vers les classes associées, en utilisant la facette **regle-ajout**.

Cette facette, nous l'avons indiqué au chapitre II.4.4.3, contient la liste des règles qui doivent être utilisées lors de l'ajout d'une valeur à l'attribut concerné. C'est à dire que cela déclenche une transmission de la forme:

```
(forall X (% # fac? 'regle-ajout)
  (X exec: %))
```

En fait, les noms des receveurs de messages (les 'X' dans notre exemple) peuvent être aussi bien des règles, que n'importe quelle entité qui répond à un message **exec**. Dans notre cas nous les utiliserons pour déclencher le mécanisme de mise en correspondance de l'entité déclenchante et de la classe à laquelle elle est reliée.

Cette connexion qui relie un attribut à la classe dont il est l'une des propriétés de définition, est réalisée lors de la création de cette classe, par la méthode **new-rep** qui lui est définie dans le type classe:

```
new-rep methode: () (forall X (% defattr ?)
  (ce type X <=fac: 'regle-ajout %))
```

De ce fait, après création de la classe 'cl-med-britan', le type **personne** devient (en supposant qu'il ne soit caractérisé que par quatre attributs seulement: l'âge, le sexe, la

profession et la nationalité) :

```
: (personne print)
(TYPE => ANIME PERSONNE
  AGE      $DOM $NOMBRE,
  SEXE     $VALEUR (MASCULIN FEMININ)
          $DEFAULT (MASCULIN),
  PROFESSION $DOM PROFESSION
          $REGLE-REQ ( CL-MED-BRITAN),
  NATION    $DOM PAYS
          $REGLE-REQ ( CL-MED-BRITAN),
)
= PERSONNE
```

création d'un individu correspondant à la définition de la classe, entraîne son ajout au sein de cette classe. Voyons quelques exemples mettant en oeuvre la classe 'cl-med-britan' :

```
; creation de deux representants ;
: (personne => peter
  age <- 32,
  profession <- medecin,
  nation <- grande-bretagne,
)
= PETER
```

```
: (personne => jean
  age <- 43,
  profession <- medecin,
  nation <- france,
)
= JEAN
```

```
: (personne allinst)
= (PETER JEAN) ; Peter et Jean sont bien des representants;
                ; du type 'personne' ;
```

```
: (cl-med-britan ?)
= PETER ; Peter est le seul membre de la classe ;
```

```
: (peter print)
(PERSONNE => PETER
  AGE      <- 32,
  NATION    <- GRANDE-BRETAGNE,
  PROFESSION <- MEDECIN,
```

```

STATUT      <- CONVENTIONNE, ; on sait que tous les medec
                                           ; Britanniques sont conventi
)
= PETER

: (cl-med-britan => james)
= JAMES ; creation d'un nouveau membre de la classe ;

: (james print)
(PERSONNE => JAMES
  PROFESSION <- MEDECIN,
  NATION     <- GRANDE-BRETAGNE,
  STATUT     <- CONVENTIONNE,
)           ; il herite de toutes les proprietes des membres;
= JAMES     ; de cette classe ;

: (cl-med-britan ?)
= (PETER JAMES) ; il a bien ete incorpore a l'extension ;
                  ; de la classe ;

: (personne allinst)
= (PETER JEAN JAMES)
; mais aussi a l'ensemble des representants de 'personne' ;

```

Ainsi, nous voyons comment il est possible de créer un représentant, soit à partir d'un type, soit à partir d'une classe, qui joue alors le rôle de prototype, pour les éléments qui en sont issus.

III.5.3 L'union des classes

Il est aussi possible, nous l'avons dit, de manipuler les classes et de les tester les unes par rapport aux autres. L'intersection et l'inclusion sont deux des traitements qu'il est possible de leur appliquer.

Unir deux classes, consiste à créer une nouvelle classe dont les éléments sont la réunion simple des éléments, mais aussi dont la définition est l'union des deux définitions.

A cet effet nous avons réalisé deux types d'inclusion: le premier ne concerne que la réunion des éléments en extension. Cette union est totalement libre, et l'attribut 'type' de la classe ainsi engendrée prend la valeur \$objet, et les attributs 'defattr' et 'defacc' sont vides. Cela signifie qu'il n'y a plus de liaison possible entre l'ensemble des entités de la base et cette classe. Cette classe devient morte.

En revanche la deuxième catégorie d'union prend en compte les définitions intensionnelles : l'attribut 'type' est calculé comme suit: soit C l'union des classes B et C, si l'une des deux classes, par exemple A, possède comme valeur de l'attribut 'type' un sur-type de l'attribut 'type' de l'autre classe, alors le type de C prend le type de A. Sinon C reçoit le type du premier sur-type commun à A et à B.

Les attributs 'defattr' et 'deffacc' doivent être eux aussi calculés: la valeur résultante est l'intersection des valeurs issues des classes originelles. Ce calcul est indispensable si l'on ne veut pas créer des illogismes: si les médecins Britanniques sont conventionnés, et non les médecins français, alors un médecin quelconque pris entre ces deux classes n'est pas nécessairement conventionné.

Par exemple:

```
: (classe => cl-med-franc
  defattr <- (profession nation),
  defacc <- (),
  type <- personne,
  profession <- medecin,
  nation <- france,
)
= CL-MED-FRANC

; union simple, sans calcul de type ou autre ;
: (classe => cl-med1 | union1 cl-med-critan cl-med-franc )
= CL-MED

: (cl-med print)
(CLASSE => CL-MED
  CELL <- (PETER JAMES JEAN)
)
= CL-MED
```

```

: (personne print)
(TYPE => ANIME PERSONNE
  AGE      $DOM $NOMBRE,
  SEXE     $VALEUR (MASCULIN FEMININ)
           $DEFAULT (MASCULIN),
  PROFESSION $DOM PROFESSION
           $REGLE-AJOUT (CL-MED-BRITAN CL-MED-FRANC),
  NATION    $DOM PAYS
           $REGLE-AJOUT (CL-MED-BRITAN CL-MED-FRANC),
)
= PERSONNE
; la nouvelle classe n'a pas de lien avec le type des objets
qu'elle contient. On dit que la classe CL-MED est morte ;

: (cl-med union cl-med-franc cl-med-britan)
= CL-MED

; Il s'agit maintenant de l'union active ;

: (cl-med print)
(CLASSE => CL-MED
  CELL      <- (PETER JAMES JEAN),
  DEFATTR   <- (PROFESSION NATION),
  DEFACC    <- NIL,
  TYPE      <- PERSONNE,
  PROFESSION <- MEDECIN,
  NATION    <- (GRANDE-BRETAGNE FRANCE),
)
= CL-MED

: (personne print)
(TYPE => ANIME PERSONNE
  AGE      $DOM $NOMBRE,
  SEXE     $VALEUR (MASCULIN FEMININ)
           $DEFAULT (MASCULIN),
  PROFESSION $DOM PROFESSION
           $REGLE-REQ (CL-MED-BRITAN CL-MED-FRANC CL-MED),
  NATION    $DOM PAYS
           $REGLE-REQ (CL-MED-BRITAN CL-MED-FRANC CL-MED),
)
= PERSONNE

; Cette classe est maintenant active ;

: (personne => marie
  age <- 29,
  sexe <- feminin,

```

```

    profession <- medecin,
    nation <- france,
  )
= MARIE

: (cl-med-franc ?)
(JEAN MARIE)

: (cl-med ?)
= (PETER JAMES JEAN MARIE)

; une classe qui n'a rien à voir : chien français ;
: (classe => chien-franc
  defattr <- (nation),
  type <- animal,
  nation <- france,
)
= CHIEN-FRANC

: (cl-franc union cl-med-franc chien-franc | print)
(CLASSE => CL-FRANC
  CELL <- (JEAN MARIE),
  DEFATTR <- (NATION),
  TYPE <- ANIME,
  NATION <- FRANCE,
)
= CL-FRANC

```

5.4 L'Intersection de classes

Pour l'intersection de deux classes, le mécanisme employé est très semblable: il existe aussi deux genres d'intersection: on qu'il y a calcul ou non des attributs 'type' 'defattr' et 'defacc'. Dans le premier cas il ne s'agit que d'une intersection des individus, et les conséquences sont identiques à celles de l'union. Dans le second cas, les valeurs des attributs 'defattr' et 'defacc' de la nouvelle classe sont la réunion de celles des classes originelles, et l'attribut 'type' est donné par le calcul: si l'une des deux classes, A par exemple, a pour type un sous-type de l'autre, alors l'attribut 'type' de C sera celui de A, sinon, on prend le plus proche

sous-type commun (s'il existe) des deux classes.

Par exemple:

```
: (classe => cl-masc
  type      <- personne,
  defattr   <- sexe,
  sexe      <- masculin,
)
= CL-MASC
; la classe des personnes de sexe masculin;

: (personne => pierre
  age       <- 53,
  profession <- tapissier,
  nation    <- france,
  sexe      <- masculin,
)
= PIERRE
; Pierre, une nouvelle personne qui n'est pas medecin ;

: (cl-masc ?)
= (PETER JEAN JAMES PIERRE)
; toutes les personnes de sexe masculin ;

: (classe => cl-med-masc | inter cl-med cl-masc)
= CL-MED-MASC

; la classe des medecins masculins est definie comme ;
; l'intersection de la classe des medecins et de      ;
; celle des etres de sexe masculin                     ;

: (cl-med-masc print)
(CLASSE => CL-MED-MASC
  CELL      <- (PETER JAMES JEAN),
  DEFATTR   <- (PROFESSION NATION SEXE),
  DEFACC     <- NIL,
  PROFESSION <- MEDECIN,
  NATION     <- (GRANDE-BRETAGNE FRANCE),
  SEXE       <- MASCULIN,
)
= CL-MED-MASC
```

```
; la visualisation de la classe des personnes montre ses ;
; liens dynamiques avec les classes ;

: (personne print)
(TYPE => ANIME PERSONNE
  AGE      $DOM $NOMBRE,
  SEXE     $VALEUR (MASCULIN FEMININ),
           $REGLE-AJOUT (CL-MED-MASC),
  PROFESSION $DOM PROFESSION,
           $REGLE-AJOUT (CL-MED-BRITAN CL-MED-FRANC CL-MED
                        CL-MED-MASC),
  NATION    $DOM PAYS
           $REGLE-AJOUT (CL-MED-BRITAN CL-MED-FRANC CL-MED
                        CL-MED-MASC),
)
= PERSONNE
```


6 La programmation par règles de production en

NG

6.1 Inférences et règles de production

Les travaux en Intelligence Artificielle ont montré l'importance des mécanismes de raisonnement dirigés par les règles. Il s'agit de la construction de systèmes dans lesquels des "filtres" (patterns), situés dans la base de connaissances déclenchent l'activation de procédures, l'assertion de prédicats, l'envoi de messages ou autres processus, ce que Waterman et F. Hayes-Roth appellent les "systèmes d'inférence dirigés par filtrage" (pattern directed inference systems). (Hayes-Roth.78)

Une catégorie importante de ces raisonnements a pu être caractérisée sous la forme de règles de production. Ces règles, sont de la forme

si antécédent alors conséquent,

sont déclenchées par chaînage avant ou arrière. Dans le premier cas, la partie 'conséquent' est activée lorsque la partie 'antécédent' correspond (match) à des informations de la base de connaissances. Inversement le déclenchement par chaînage arrière tend à déterminer si le 'conséquent' d'une règle peut être prouvée compte tenu des informations et des autres règles disposées dans la base.

L'organisation d'un tel système comprend ordinairement trois parties: la base de faits, les règles qui agissent sur la base et le moteur d'inférence qui contrôle l'activation des règles.

Dans la majorité des systèmes développés, la structure des données de la base est souvent très élémentaire: prédicats, réseaux sémantiques simples, etc... Récemment quelques chercheurs ont proposé une approche plus élaborée qui consiste à appliquer des systèmes de règles de production à des structures plus complexes, schémas ou "frames", afin d'organiser au mieux l'espace des connaissances de base (AM (Lenat.82), CENTAUR (Aikins.83), TOUBIB (Fargues.83),...), et d'obtenir plus de puissance descriptive, déductive et inductive lors de manipulation de domaines "réels".

Par rapport aux systèmes de règles de production "purs", qui n'emploient que des prédicats, l'utilisation de schémas permet de spécifier des valeurs par défaut ou des contraintes, mais surtout de définir des démonstrateurs, procédures activées lors du placement d'une valeur, ou des attentes, c'est à dire des demandes d'information supplémentaires que le système se charge de lui trouver en déclenchant des procédures particulières.

Inversement, pour les concepteurs de systèmes de schémas, les règles d'inférences fournissent un moyen pratique de préciser les raisonnements et les traitements à effectuer sur la base conceptuelle. Cette démarche s'avère non seulement plus élégante que le simple emploi de procédures, mais aussi plus efficace en termes de contrôle et d'explication des mécanismes de raisonnement.

De nombreux langages d'Intelligence Artificielle destinés à la représentation et à la manipulation de connaissances comportent un mécanisme de filtrage. Qu'il s'agisse de PLANNER (Hewitt.71), de KRL (Bobrow.77-b), de AIMDS (Sridharan.78), etc., ce type de contrôle a souvent été l'un des moteurs essentiels de la construction de tels langages, et le récent développement des systèmes experts a montré de manière encore plus critique son importance.

Ceux-ci ont transformé le point de vue que l'on avait sur les mécanismes d'inférence. Au départ simples théorèmes servant à la résolution de problèmes logiques, ils sont devenus les mécanismes de raisonnement de base d'un grand nombre de systèmes informatiques liés à l'Intelligence Artificielle.

A l'origine, avec un langage comme PLANNER, l'inférence

était considérée comme une procédure agissant sur une base de données, laquelle était constituée d'un ensemble d'assertions représentées comme des n-tuples LISP :

(SUR BLOCK1 TABLE)

(DESIRE TOTO (MANGER TOTO CONFITURE))

A partir de ce langage, considéré par l'ensemble de la communauté de l'I.A comme une étape essentielle des recherches en représentation des connaissances, de nombreuses voies furent défrichées, qui portèrent notamment sur les points suivants:

- Les inférences doivent-elles être conçues comme des entités procédurales ou comme des déclarations qu'un interpréteur extérieur doit gérer?
- Quels doivent être les mécanismes d'inférence de base: chaînage avant, arrière, résolution logique, utilisation de démons, etc...
- La structure des assertions doit-elle être fixe ou très générale?

Les tenants de la première approche estiment que des prédicats à structure fixe (c'est à dire dont le nombre et la fonction des arguments fait partie intégrante du système) sont suffisant pour décrire des connaissances générales. En effet, dans la plupart des systèmes experts, les assertions se présentent sous la forme de quadruplets

<objet, attribut, comparateur, valeur>

parfois assortis de pondérations de crédibilité (MYCIN (Shortliffe.76), PROSPECTOR (Konolige.79),...).

Les partenaire de la deuxième considérèrent qu'il fallait réaliser des systèmes plus généraux, et pas se restreindre à des formes informatives trop élémentaires. Cette démarche conduisit au développement de langages déclaratifs tels que PROLOG

(Colmerauer.82), ou le récent SNARK (Laurière.82).

En réalité, la forme des assertions relève d'une question plus générale qui a été soulevée dans le chapitre I et qui concerne le formalisme employé pour la représentation des concepts.

Les prédicats à structure fixe s'insèrent très bien dans un "formalisme objet", où les unités conceptuelles sont décrites comme une liste d'attributs. Pour s'en convaincre, il suffit de considérer la plupart des systèmes experts. Qu'il s'agisse de EMYCIN, de PROSPECTOR ou de plus récents comme CENTAUR ou TOUBIB, tous utilisent, comme support de leurs informations factuelles, de leurs assertions, une structure de donnée à caractère "objet": arbre contextuel pour EMYCIN, réseaux sémantiques pour PROSPECTOR, 'schémas' pour CENTAUR et TOUBIB.

En revanche les prédicats à structure quelconque sont caractéristiques d'un formalisme relationnel, où le domaine de la connaissance est décrit en termes de relations entre objets, lesquels n'ont d'existence qu'à travers ces relations.

Nous avons voulu montrer comment il est possible d'employer des règles de production intégrées dans un formalisme "orienté objet", et d'être ainsi à même de fournir un environnement de programmation plus riche pour la réalisation d'applications dans le domaine de la représentation des connaissances.

Nous ne voulions pas tomber dans le piège qui consiste à créer un moteur d'inférence séparé du système de représentation. Bien que cette approche soient utilisés dans certains systèmes tels que TOUBIB, nous ne pensons pas que cette démarche soit la bonne, car elle suppose une incohérence conceptuelle (ou plutôt méta-conceptuelle, puisqu'il s'agit du formalisme utilisé pour la structure des concepts) en distinguant d'une part un système de représentation "orienté objet" et d'autre part un moteur d'inférence utilisant un formalisme relationnel.

Au contraire notre démarche se veut intégrante, pour pouvoir produire les outils les plus efficaces, et les plus adaptés possible au désir du programmeur. La programmation est un acte de création et d'expression: encore faut il avoir les

ens de s'exprimer. Cela ne peut se faire que dans la
ersité des démarches et la confrontation des points de vue:
l un environnement de programmation suffisamment ouvert peut
e à même de fournir le dynamisme nécessaire à une telle
ne.

6.2 Utiliser des règles de production en MERING

Afin d'être à même d'exploiter toutes les ressources de
programmation par règles de production, nous avons intégré en
NG, un mécanisme qui permet d'exprimer les connaissances de
ère déclarative, tout en conservant l'environnement de
rammation de MERING.

Il s'agit en fait de fournir au programmeur un
ronnement de programmation optimum, et de lui laisser le
x entre différentes approches.

Ces mécanismes de manipulation de règles de production ne
ent pas être considérés comme des accessoires ajoutés à
ifice principal. Au contraire, il s'agit d'une véritable
gration, d'une augmentation du langage.

En effet, ces règles sont des entités MERING, issues d'un
générique **règle**, et manipulées grâce à l'envois de
ges. Elles peuvent être organisées à l'intérieur de
mes de production (intitulés **paquets** de règles) qui
fient les règles, leur déclenchement et les mécanismes de
ôle du filtrage qui y sont associés. (la définition des
ts est décrite au chapitre III.6.6

Elles voient leur utilisation totalement incorporée au sein
angage. Trois mécanismes d'activation sont disponibles: par
loi d'autres règles, par transmission de type comportement,
mplement par déclenchement de réflexes.

Les règles peuvent être activées individuellement, ou par l'intermédiaire des paquets, en utilisant les mécanismes de contrôle qui y sont associés. Une règle peut appartenir à plusieurs paquets différents, et donc être déclenchée par des mécanismes différents.

L'emploi de règles de production permet de constituer aisément un système déductif complet (un système expert, par exemple), ou un système d'élaboration de plans pour robots.

Une fois définies, ces règles sont compilées sous forme de transmission MERING. De ce fait leur utilisation ne pénalise pas les performances générales du système. La compilation des règles s'effectue seulement si le besoin s'en fait sentir, c'est à dire lors de sa première utilisation. De cette manière, une règle qui n'est pas utilisée ne sera pas compilée et donc ne prendra pas plus de place en mémoire que n'en requiert sa définition.

Les règles en MERING correspondent à des énoncés du type:

si <condition> **alors** <action>

où la partie condition est formée d'une suite logique (c'est à dire gouvernée par des opérateurs logiques **et**(and) et **ou** (or)) de transmissions élémentaires qui contiennent les prémisses de la règle, et où la partie action se compose d'une ou de plusieurs actions à exécuter lorsque les prémisses sont satisfaites par les données de la base.

Ces actions peuvent être absolument quelconques: affectation d'une valeur, création d'un représentant, impression d'un résultat, déclenchement d'un comportement, etc... (Nous examinerons cela plus loin, avec des exemples).

Les règles peuvent contenir des variables: à ce sujet, deux types de variables sont proposés: les arguments et les variables proprement dites.

Les **arguments** sont des entités virtuelles, qui prendront une valeur réelle lors de l'appel de cette règle, c'est à dire lors de son déclenchement.

Les **variables** proprement dites (et que nous appellerons dorénavant "variable" tout court) peuvent prendre un domaine qui est: soit une liste de valeurs explicites soit un ensemble d'entités appartenant à un type déterminé. Elles sont liées par un quantificateur, lequel aura pour objet de contrôler le mécanisme d'unification de la règle.

Par exemple décrivent une règle qui représente la relation de parenté 'oncle': en langage ordinaire, les oncles sont les frères du père, ou les frères de la mère. Cette règle s'écrit en MERING:

```
(regle => oncle
  args  :: X,
  soit  :: (Y Z),
  dom   :: (personne personne),
  quant :: (U U),
  condit :: (or
    (and
      (X pere = Y)
      (Y frere = Z))
    (and
      (X mere = Y)
      (Y frere = Z))),
  action :: (X oncle <= Z),
; lorsqu'on trouve un oncle on le place sur ;
; l'attribut 'oncle' de l'argument. ;
)
```

L'utilisation de la règle est effectuée en lui envoyant un message qui définit l'ensemble de la base sur lequel la règle doit être appliquée. Par exemple si l'on désire connaître tous les oncles de TOTO, il suffit de transmettre:

```
: (oncle exec: toto)
= ONCLE
```

et l'attribut oncle de toto contient maintenant la liste de ses

oncles:

```
: (toto oncle ?)
= (JEAN PAUL)
```

Si l'on désire appliquer la règle sur tout un ensemble d'entité, on emploiera le message **execall**: de la manière suivante:

```
(oncle execall: '(toto riri fifi))
```

S'il s'agit de tous les représentants d'une entité, par exemple si l'on veut connaître tous les oncles de toutes les personnes, on pourra envoyer le message:

```
(oncle exetype: personne)
```

transmission équivalente de

```
(oncle execall: (personne allinst))
```

Il sera ensuite possible de placer le déclenchement de cette règle sur un attribut en utilisant un réflexe. Pour obtenir tous les oncles d'une personne, on pourra placer cette règle sur la facette '\$si-req' de l'attribut 'oncle' du type personne. De cette manière dès que l'on aura besoin de cette valeur, la règle sera activée par réflexe, en chaînage arrière.

Mais cette utilisation n'est pas la seule. Imaginons que nous voulions déclencher cette règle en chaînage avant (par exemple pour connaître immédiatement tous les oncles possibles lors de l'affectation du père et de la mère de la personne en question). Il suffit alors de placer l'activation de cette règle sur la facette '\$si-ajout'.

Voici le type 'personne' avec ses relations de parenté:

```
(type => $objet personne
  age    $dom $nombre
  profession $dom profession,
```

```
pere $dom personne
      $si-ajout (oncle exec: %),
mere  $dom personne
      $si-ajout (oncle exec: %),
oncle $dom personne
      $si-req (oncle exec: %),
)
```

Les déclenchements d'une règle par chaînage avant ou arrière sont tellement courants que des facettes ont été dédiées à cet effet. Il s'agit des facettes **regle-ajout** **regle-lu** et **regle-req** qui correspondent respectivement à des activations de type **\$si-ajout**, **\$si-lu** et **\$si-req**. Dans ces facettes, il n'est besoin que d'indiquer le nom de la règle, l'activation étant réalisée automatiquement.

En effet, les actions d'ajout (ou d'affectation) et de lecture déclenchent automatiquement l'une de ces trois transmissions:

Lors de l'ajout ou de l'affectation d'une valeur:

```
(forall X (% # fac? '$regle-ajout)
  (X exec: %))
```

Lors de la lecture d'une valeur:

```
(forall X (% # fac? '$regle-lu)
  (X exec: %))
```

ou

```
(forall X (% # fac? '$regle-req)
  (X exec: %))
```

Le type **personne**, écrit avec ces facettes, se présente ainsi:

```
(type => $objet personne
  age      $dom $nombre,
  profession $dom $nombre,
  pere     $dom personne
           $regle-ajout oncle,
  mere     $dom personne
           $regle-ajout oncle,
  oncle    $dom personne
           $regle-req oncle,
)
```

En ce qui concerne les critères d'utilisation, et le mode de déclenchement de ces facettes, on pourra se reporter au chapitre IV.

III.6.3 Définition d'une règle

L'exemple précédent nous a permis de voir la structure générale d'une règle. En effet, MERING permet d'introduire les règles de production comme des entités de la base, le système se chargeant ensuite d'engendrer la transmission correspondante ainsi que ses modes d'accès.

En MERING, décrire une règle consiste à créer un représentant du type générique **règle**, sous la forme suivante:

```
(regle => ... ; nom de la règle
  args  :: ... ; arguments passés lors du déclenchement
  soit  :: ... ; liste des variables
  dom   :: ... ; domaine de variation des variables
  quant :: ... ; quantificateurs des variables
  condit :: ... ; prémisses
  action :: ... ; conséquent
)
```

Chacun des attributs de l'entité de type 'regle' contient des informations servant à la compilation de l'inférence.

- **args** comprend les variables arguments, c'est à dire les entités virtuelles qui seront actualisées lors du déclenchement de la règle.
- **soit** contient la liste des variables qui sont utilisées pour la définition de l'inférence. L'ordre des arguments a son importance. Lors de la compilation de la définition sous forme de transmission MERING, celle ci prend la forme de boucles imbriquées, chaque boucle étant associée à une variable et à un domaine de variation.

La variable située le plus à gauche correspond à la boucle interne, et la variable placée à l'aile droite constituera la boucle externe. Tenir compte de cet effet peut sensiblement accélérer le processus de traitement.

- **dom** contient le domaine de variation des variables, c'est à dire le type des entités substituées aux variables. Par exemple si X représente une personne, le domaine de variation est le type 'personne'. Par défaut le domaine de variation est \$objet, c'est à dire que les opérations de recherches sont effectuées sur l'ensemble des informations "conceptuelles" de la base (en excluant donc toutes les entités "système" telles que séquences, classes ou ... règles).
- **quant** définit les quantificateurs applicables à chacune des variables. Deux quantificateurs seulement ont été définis en MERING (logique non-modale): il s'agit des quantificateurs universels et existentiels bien connus, et sont représentés respectivement par la lettre 'U' et 'E'.

Employer un quantificateur universel, indique à la règle de tenter de s'appliquer sur tous les représentants de la base, que la règle s'applique ou

non. Le quantificateur existenciel en revanche demande à la règle de cesser d'être activée si elle réussit à s'appliquer à l'un des éléments de ce domaine. Par défaut le quantificateur est universel.

- **condit et action** contiennent les parties antécédent et conséquent de l'inférence. La première est constituée d'une liste de transmissions élémentaires reliées par des opérateurs logiques telles que **and** et **or**, alors que la seconde est une transmission quelconque qui sera évaluée si les prémisses sont satisfaites.

III.6.4 Interêt des règles de production

L'intégration de règles de production au sein d'un système d'acteur, et leur compilation sous formes de transmissions, présente de nombreux avantages par rapport à l'emploi d'un moteur d'inférence associé à une structure de donnée, comme c'est le cas dans la plupart des systèmes experts (MYCIN, PROSPECTOR, CENTAUR, TOUBIB,...):

- **Uniformité:** Le modèle MERING reste intact. Il n'y a toujours qu'un système, qu'un langage, puisque les règles sont compilées dans les termes du langage. Cette présentation a aussi vertu de cohérence. Lorsqu'on utilise deux systèmes, un moteur d'inférences pour les règles et un langage de schémas (frames) pour les données, le contrôle général doit être laissé aux schémas (aux structures de données en général) ou au moteur d'inférences.

L'utilisation d'une telle intégration, supprime ces possibilités de tensions, et assigne le contrôle aux éléments désirés, suivant la volonté du programmeur qui travaille toujours à l'intérieur du même langage. Cette caractéristique le désigne comme un noyau de système expert moins catégorique quant à la structure de contrôle que des systèmes comme CENTAUR

ou TOUBIB.

Simplicité: Les règles sont implantées sous la forme de transmissions. Elles n'ont de ce fait pas besoin d'un mécanisme d'unification à une base de faits. Elles "sont" leur propre interpréteur. L'écriture d'un moteur d'inférence devient inutile: ou bien le programmeur explicite leur activation, en envoyant un message qui déclenche directement la règle (ou le paquet de règles) à appliquer, ou bien il laisse cette tâche aux réflexes, les données placées dans les attributs actionnant automatiquement les règles adéquates.

Dans ce cas, nous sommes donc en présence d'un véritable "système d'inférences dirigées par les données". Les filtres (ou "patterns") n'ont pas à être unifiés à des données. C'est l'ajout ou la lecture d'une information, d'un attribut, qui déclenche l'inférence désirée, par chaînage avant ou arrière.

Généralité: Lorsqu'elles sont activées par réflexes, les règles de production sont employées aussi bien en chaînage avant qu'arrière. Il n'y a pas de définition a priori du sens d'utilisation, qui ne dépend que de l'opération effectuées sur les données. Lors de l'insertion de nouvelles informations, les règles sont déclenchées par chaînage avant. Inversement la recherche d'informations amène le système à fonctionner en chaînage arrière.

Modularité: Les systèmes à règles de production tendent à se structurer. Les efforts de certains systèmes experts pour utiliser des schémas sont là pour en témoigner. Non seulement nous utilisons la modularité inhérente à notre modèle "orienté objet", qui différencie entre types et représentants et qui utilise un mécanisme d'héritage multiple pour la structuration des connaissances, mais de plus il est possible d'organiser les règles en paquets, et donc de regrouper les inférences par leur fonction.

-Indépendance fonctionnelle: L'indépendance des règles par rapport aux faits ou aux autres règles, qui est une condition de bon fonctionnement des systèmes à règles de production, est conservée en Mering, malgré la forte structuration des connaissances. Une règle peut être ajoutée ou enlevée d'un paquet sans remettre en cause l'organisation des informations dans la base.

A ce sujet, il est bon de signaler la différence qui existe entre indépendance structurelle et indépendance fonctionnelle (Rady.83). La première estime que du niveau superficiel jusqu'aux étages d'implantations, les informations doivent être séparées de l'interpréteur qui les manipulent. Il s'agit là de la vision "déclarative" classique qui dissocie entièrement les assertions du module de gestion de ces assertions.

En revanche, la seconde est moins radicale. Elle suppose simplement que l'adjonction et la suppression de règles d'inférences ne doit pas remettre en cause la structure de la base, et entraîner une réécriture d'autres règles introduites préalablement.

L'indépendance fonctionnelle est une des caractéristiques de Mering. Les règles ajoutées sont compilées et insérées automatiquement dans la structure de la base, et leur suppression relève d'une action aussi mécanique, puisqu'elle consiste à effacer la définition de la règle et à éliminer sa présence des paquets de règle. Ce processus est totalement transparent pour l'utilisateur, qui ne touche pas plus à la structure des connaissances établies précédemment que dans un système déclaratif pur.

-Performance: Du fait de cette transformation des règles en actions procédurales, l'explosion combinatoire, et le temps de traitement en croissance exponentielle avec le nombre de règles et de faits

contenus dans la base est moins à craindre.

Une illustration du mécanisme d'activation des règles

Afin de montrer comment s'applique une règle, nous reprendrons notre jeu d'aventure, et nous supposons que Galark, notre guerrier, doit franchir des obstacles, tels qu'ouvrir une porte, ou éclairer un boyau obscur.

Encore faut il qu'il en ait la capacité. Pour ce faire, une règle de production, **capac** gouverne le fait d'être à même ou non d'accomplir une action. En langage ordinaire, on dira que X est capable d'exécuter une action Y, s'il possède dans son équipement un objet Z qui sert à Y. La définition de la règle (et des nouveaux éléments de la base du jeu d'aventure) est la suivante:

```
(regle => capac
  args :: (X Y),
  soit  :: Z,
  dom   :: objet,
  quant :: E,
  condit :: (and
    (not (capable == Y))
    (X equipt == Z)
    (Z sert-a == Y)),
  action :: (X capable-de <= Y),
)

(lanterne sert-a <- eclairer)
(clef sert-a <- ouvrir-porte)
```

Ici nous placerons le déclenchement de la règle dans une méthode associée au type personnage dont Galark est issu. L'ajout d'une méthode à un type est facilité par l'applicateur **<-meth** qui prend en argument le corps de définition d'une méthode.

Cette méthode stipule que si le sujet ne comporte pas la capacité V dans son attribut **capable**, alors il active la règle **capac**, puis il relit cet attribut.

```
(personnage capable <-meth| (V) (if (% capable == V)
                                T
                                (capac exec| % V & % capable == V
```

Et voici comment se comporte la règle:

```
: (galark equipt <= lanterne)
= GALARK
```

```
: (galark capable eclairer)
= T
```

```
: (galark capable ouvrir-porte)
= NIL
```

```
: (galark equipt <= clef | capable ouvrir-porte)
= T
```

6.5 Les paquets de règles

Nous l'avons dit, les règles peuvent être regroupées sous forme de paquets de règles, qui constituent de véritables thèmes de production. Cette présentation s'avère nécessaire lorsque plusieurs règles doivent être testées, et que l'on ne peut pas expliciter le déclenchement de chacune d'entre elles.

Ces paquets sont activables par des messages identiques à ceux qui s'appliquent à une règle seule. Evidemment, le receveur de message étant différent (il s'agit ici d'un paquet et non pas d'une règle), le résultat sera différent.

Par exemple le message **exec:** envoyé à un paquet, déclenchera successivement l'ensemble des règles du paquet avant le type de contrôle désiré.

Le mécanisme de déclenchement des règles est programmable: il est possible ainsi de n'appliquer qu'une seule fois l'ensemble d'un paquet, ou de le répéter tant que cela est possible, ou bien encore d'essayer toutes les règles jusqu'à ce que l'une d'entre elles s'applique.

Les modalités d'activation sont contrôlées par quelques attributs:

boucle indique si l'ensemble des règles doit être examiné une seule fois (valeur NIL) ou bien tant qu'il est possible d'appliquer l'une d'entre elles (valeur T).

répète précise le nombre de tours qui peut être réalisé (pour éviter de boucler pour toujours), si l'attribut 'boucle' est T. Une valeur NIL ou 0 indique qu'il n'y a pas de contraintes sur le bouclage du système. (valeur par défaut : 10).

- **toutes** signale si toutes les règles doivent être testées (valeur T) ou bien si les opérations doivent être arrêtées (ou recommencées depuis le début si l'attribut 'boucle' est T) dès qu'une règle est applicable sur la base.
- **trace, detrace et trace?** sont trois méthodes dont la fonction est de : commander un traçage des règles lors de leur application, supprimer cette trace ou signaler si la trace est en marche.

un exemple d'application des paquets de règles

L'exemple suivant montre comment un système de règles peut être utilisé pour la construction de plans par un robot dont la fonction est, ce qui est banal pour un robot, d'empiler et de dépiler des boîtes.

Il s'agit, à partir d'une situation initiale décrite par un certain nombre d'arrangements des boîtes, de déterminer les actions à accomplir afin d'obtenir une situation finale, un autre arrangements des boîtes.

Cet exemple, classique dans les systèmes déductifs fondés sur la résolution logique, ou plus généralement pour tous ceux qui sont fondés sur le "formalisme relationnel", permet de montrer comment il est possible de formuler aisément la description d'un tel problème en MERING, malgré son attachement au "formalisme objet".

Les boîtes sont des objets MERING, qui peuvent être placées sur la table (attribut surtable prend la valeur T) ou sur un autre bloc: une boîte A située sur une autre boîte B, possède dans son attribut 'sur' la valeur B. Une boîte située au sommet d'une pile voit son attribut 'clair' prendre la valeur T.

Les méthodes **sur** et **surtable** servent à déclencher les règles d'empilage et de dépilage des cubes. Par exemple la transmission

(A sur C)

tente de parvenir dans l'état où le but A est sur le bloc C donc à empiler la boîte A sur la boîte C, en essayant d'appliquer l'une des règles du paquet correspondant.

Le paquet **depile** contient les règles qui permettent de poser (si c'est possible) la boîte passée en argument à terre.

Le paquet **empile** se charge de placer (si c'est possible) la première boîte passée en argument sur la seconde.

Ces paquets ont comme modalité de contrôle les spécifications par défaut, c'est à dire que les attributs 'boucle', 'toutes' et 'répète' prennent la valeur nulle.

Pour parvenir à leur résultat, les règles s'appellent mutuellement, c'est à dire qu'elles invoquent les paquets 'empile' ou 'depile' (résultats des transmission X SUR Y et X SURTABLE) récursivement.

; Base de démonstration des règles sur un monde de boîtes ;

```
(type => $objet boîte
  surtable methode: () (depile exec: % & T),
  sur      methode: (V) (empile exec: % V & T),
  sur      $dom $boîte,
  surtable $dom $bool,
  clair    $dom $bool,
)
```

; creation des objets et situation initiale ;

```
(boite => A
  surtable <- T,
)
```

```
(boite => B
  sur <- A
)
```

```
(boite => C
  sur <- B,
  clair <- T,
)
```

; la base des regles ;

```
(regle => depil0
  args :: X,
  condit :: (X surtable ?),
  action :: T,
)
```

```
(regle => depil1
  args :: X,
  soit :: Y,
  dom :: boite,
  quant :: E,
  condit :: (and
    (X clair ?)
    (X sur = Y)),
  action :: (X surtable <- T
    & Y clair <- T
    & X sur <- ()
    & print "depilage de :" X ),
)
```

```
(regle => depil2
  args :: X,
  soit :: Y,
  dom :: boite,
  quant :: E,
  condit :: (and
    (not (X clair ?))
    (Y sur = X)),
  action :: (Y surtable & X surtable),
)
```

```
(regle => empil0
  args  :: (X Y),
  condit :: (and
              (not (X sur = Y))
              (X surtable ?)
              (X clair ?)
              (Y clair ?)),
  action :: (X surtable <- ()
            & Y clair <- ()
            & X sur <- Y
            & print "empilage de " X "sur" Y ),
)
```

```
(regle => empil1
  args  :: (X Y),
  soit  :: Z,
  dom   :: boite,
  quant :: E,
  condit :: (and
              (not (X sur = Y))
              (X surtable ?)
              (X sur = Z)
              (X clair ?)
              (Y clair ?)),
  action :: (Y clair <- ()
            & X sur <- ()
            & X sur = Y
            & Z clair <- T
            & print "empilage de " X " sur " Y ),
)
```

```
(regle => empil2
  args  :: (X Y),
  soit  :: Z,
  dom   :: boite,
  quant :: E,
  condit :: (and
              (not (X sur = Y))
              (Y clair ?)
              (not (X clair ?))
              (Z sur = X)),
  action :: (Z surtable & X sur Y),
)
```

```
(regle => empil3
  args  :: (X Y),
  soit  :: Z,
```

```

dom      :: boîte,
quant    :: E,
condit   :: (and
              (not (X sur = Y))
              (not (Y clair ?))
              (X clair ?)
              (Z sur = Y)),
action   :: (Z surtable & X sur Y),
)

```

```

(regle => empil4
  args    :: (X Y),
  soit    :: Z,
  dom     :: boîte,
  quant   :: E,
  condit  :: (and
              (not (X sur = Y))
              (not (Y clair ?))
              (not (X clair ?))
              (Z sur = X)),
  action  :: (Z surtable & X sur Y),
)

```

```

; les paquets      ;

```

```

(paquet => depile | <- '(depil0 depil1 depil2) )

```

```

(paquet => empile | <- '(empil0 empil1 empil2 empil3))

```

```

; situation finale et déclenchement ;

```

```

: (B surtable & C sur B & A sur C)

```

```

"depilage de " C

```

```

"depilage de " B

```

```

"empilage de " C " sur " B

```

```

"empilage de " A " sur " C

```

```

T

```

Définition du type paquet

Nous donnons ci-dessous la description du type paquet. Afin de simplifier son expression, nous n'avons donné que la version par défaut des comportements d'activation des règles, c'est à dire celle qui est exécutée lorsque les paramètres de contrôle ont leur valeur par défaut.

```
(type => seq paquet
  boucle   $dom   $bool,
  toutes   $dom   $bool,
  repete    $dom   $nombre
           $default 10,
  trace    methode: () (forall X (% ?) (X $trace <- T)
                        & $trace <- T),
  detrace   methode: () (forall X (% ?) (X $trace <- NIL)
                        & $trace <- NIL),
  trace?    methode: () (% $trace ?),
  exec:     methode: (-arg-) (exist -R- (% ?)
                        (-R- exec: -arg-)),
  exec|     methode| (-larg-) (exist -R- (% ?)
                        [-R- 'exec| -larg-] ),
  execall:  methode: (-arg-) (exist -R- (% ?)
                        (-R- execall: -arg-)),
  exetype:  methode: (-arg-) (exist -R- (% ?)
                        (-R- exetype: -arg-)),
)
```

III.6.6 Description d'un analyseur de langage naturel en MERING

Afin d'appliquer le langage MERING à un exemple de taille importante, nous avons testé la possibilité de traduire l'analyseur déterministe de langage naturel de M.RADY (Rady.83) en MERING.

L'objectif de cette opération de réécriture est double:

- 1) Développer un analyseur intégré avec un système de représentation, afin de disposer d'un outil adéquat pour poursuivre des recherches sur des systèmes d'analyse qui fusionnent tous les processus de traitement: syntaxique, sémantique, pragmatique, inférentiel et contextuel.
- 2) Montrer l'intérêt de MERING pour la réalisation de projets d'envergure.

Les caractéristiques principales de cet analyseur sont les suivantes:

- 1) le processus d'analyse est déterministe. Cela signifie qu'il n'effectue ni retour en arrière, ni traitement en parallèle. Il est fondé sur le regard en avant et le développement quasi-simultané des constituants de la phrase.
- 2) le mécanisme de traitement prend la forme d'un ensemble de règles de production, qui décrivent à la fois les informations nécessaires à l'analyse (syntaxe, sémantique, pragmatique,...) et le processus de contrôle de ces règles. Ces règles sont organisées sous la forme de paquets de règles
- 3) L'analyseur ne sépare pas les données syntaxiques des informations sémantiques.
- 4) Le résultat de l'analyse est donné sous la forme d'une structure interne qui exprime le sens de la phrase.
- 5) L'analyseur est indépendant de la langue et du système de représentation des connaissances (à condition qu'il soit lié à un "formalisme objet"). Toutes les données étant mises sous la forme de règles, il suffit de réécrire ces règles pour adapter l'analyseur à une

situation, un système de représentation ou un langage différent.

Dans l'analyseur original de RADY, les règles sont écrites l'aide d'un langage formel, LERTOL (Langage d'Ecriture des les pour le Traitements et Opérations Linguistiques), qui met d'exprimer sous une forme proche du Français les rations de traitement utilisées à l'intérieur des règles, et à dire les conditions et les actions.

Notre travail a porté essentiellement sur la traduction de TOL en MERING, avec les exigences suivantes:

Le système doit être intégré à MERING. Il ne s'agit pas de créer un interprète de LERTOL (ni un compilateur) mais d'utiliser directement la richesse d'environnement de MERING, pour y représenter directement les règles de traitement de l'analyseur.

La syntaxe de l'écriture des règles doit être celle de MERING, pour éviter à un utilisateur d'avoir à apprendre une autre syntaxe.

Les règles résultantes doivent être aussi proche que possible de leur modèle initial, et la structure de l'analyseur ne doit pas être modifiée, afin de limiter les effets de bords indésirables et de conserver les résultats de l'étude de M.RADY.

Voici deux règles écrites en LERTOL, et leur représentation MERING.

1) règle d'interruption au niveau propositionnel.

en LERTOL:

```
[engendre_gp;pact,nact,vact;10.
variable:S1;S2;S3.
var1 a_les_traits 'prep' et
var2 a_les_traits 'gn' -->
insere nouv structure avant var1;
marque S0 'gp';
adjoins ds(var2) a S0;
attache var1 a S0;
attache var2 a S0;
si var2 a_les_traits 'quid' alors marque S0 'quid qu_comp';
si var2 est_termine alors termine S0
sinon detache;]
```

en MERING:

```
(reg-an => engendre-gp
  paquets :: (pact nact vact),
  prior   :: 10,
  variable :: (var1 var2),
  condit   :: (and
    (var1 a-traits 'prep)
    (var2 a-traits 'gn')),
  action   :: (struct => | insere-av var1
    & S0 marque 'gp
    & S0 adjoins (var2 ds ?)
    & S0 attache var1
    & S0 attache var2
    & if (var2 a-traits 'quid)
      (S0 marque '(quid qu-comp))
    & if (var2 est-termine)
      (S0 termine)
    (detache) ),
)
```

2) Déterminer le type de la proposition:

en LERTOL:

```
[quest_quid;prop[type.
S1 a_les_traits 'quid' et
S1 a_un_des_traits 'gn gp' -->
marque S0 'quest quest_quid domine_comp';
marque S1 'comp';
attribue 3 a ds(S0) type ass.moti;
attache S1 a S0;
fait-referer ds(S0) type ass.motif a ds(S1);
desactive 'prop_type';
active 'contenue_quid';]
```

et en MERING

```
(reg-an => quest-quid
  paquets :: prop-type
  condit  :: (and
    (S1 a-traits 'quid)
    (S1 a-un-trait '(gn gp))),
  action  :: (S0 marque '(quest quest-quid domine-comp)
    & S1 marque 'comp
    & S0 ds ? | motif <- 3
    & S0 attache S1
    & S1 := (S0 ds ? | motif ?)
    & prop-type desactive
    & contenue-quid active),
)
```

Cette traduction a été rendue possible grâce à la présence de caractéristiques indispensables à l'analyseur et déjà présentes en MERING, telles que la notion de règle de production et celle de paquet de règle.

Les règles de traitement sont simplement des entités MERING issue d'un sous-type de **regle** : **reg-an**. La seule différence avec 'règle' concerne le traitement des priorités.

Les paquets de règles sont eux aussi issus d'un sous-type du type **paquet** initial: **paquet-an**. Il dispose en plus d'un mécanisme de réorganisation des règles en fonction de leur priorité. Le mode de déclenchement est standard (option 'toutes' et 'boucle' à NIL).

De plus ces paquets comporte un certain nombre de méthodes

telles que 'active', 'desactive', 'detache' et 'reprend' qui permettent d'accomplir des opérations sous un mode de co-routines.

Les structures engendrées et manipulées au cours de l'analyse sont des entités issues du type **struct** qui dispose de nombreuses méthodes qui servent lors du traitement.

```
(type => $objet struct
  traits $dom,
  fils   $dom struct,
  pere   $dom struct,
  ds     $som descr-sem,

; les méthodes ;
  fils  methode: (LT) ; ramène le fils de % qui possède ;
                        ; les traits déterminés par LT ;
  pere  methode: (LT) ; ramène le père de % ;
  ascendant methode: (LT) ; ramene l'ascendant ;
;   etc ... ;
)
```

Les descriptions sémantiques contiennent les éléments signifiants qui sont utilisés et engendrés au cours de l'analyse: les 'assertions', qui correspondent à l'organisation thématiques des phrases, les 'concepts' qui sont des prédicats à structure de cas (Sabah.78), et les 'objets' qui représentent toutes les informations non prédictives. Il faut signaler que ces notions pourraient très aisément évoluer dans un environnement différent que celui qu'a imaginé RADY dans son étude.

Leur structure est la suivante:

```
(type => $objet descr-sem
; les méthodes attachées aux descriptions sémantiques ;
  a-traits  methode: (LT) ; identique à a_les_traits de LERTOL ;
  a-un-trait methode: (LT) ; identique à a_un_des_traits ;
  a-ql-trait methode: (LT) ; identique à a_quelques_uns_des_traits;
; etc... ;
)
```

Pour la description des types **assertion**, **concept** et **objet** qui sont des sous-types de **descr-sem**, nous avons directement repris la définition qu'en donne RADY dans sa thèse.

Conclusion

Nous avons voulu démontrer la possibilité de traduire un système complet écrit dans un autre langage (en l'occurrence IMULA) en MERING, en intégrant cette application comme une extension du langage lui-même.

La traduction revient alors à écrire quelques types, leurs attributs et leurs comportements, puis à représenter les objets informatiques de l'application sous cette forme.

Elle est facilitée par le caractère ouvert de MERING, qui est à la fois un langage de représentation et de manipulation d'entités, au sens le plus large du terme.

Nous avons comme projet de finir cette implantation de l'analyseur déterministe de RADY en MERING, et de le présenter comme une extension du langage. Il servira alors d'outil de base pour la construction de systèmes de compréhension de langage naturel.

Le regroupement des éléments et des procédures de traitement sous la forme d'entités permet de réaliser aisément des systèmes d'analyse de langage naturel comme l'a montré NOGRAD (Winograd.83).

Il est dans nos projets de réaliser quelques analyseurs standards (ATN, grammaires de cas, etc...) de manière à fournir un environnement adéquat au développement de systèmes d'analyse récits (Grau.83) ou de gestion de dialogue en langage naturel.

CHAPITRE IV

Implantation et manuel de référence du langage

Ce chapitre traite des aspects techniques du langage MERING, de sa structure d'implantation actuelle et des différents choix qui ont été réalisés lors du développement de l'interpréteur.

De plus nous indiquons les entités et comportements qui sont à la base du système et sans lesquels il n'est pas possible de travailler sérieusement.

La version de MERING qui est proposée ici n'est pas une version définitive, mais elle utilise tous les concepts que nous avons développés dans le chapitre II, et les exemples du chapitre III ont tous été écrits avec cette implantation.

IV.1 Structure de l'interpréteur MERING

L'interpréteur MERING que nous avons réalisé doit être considéré comme un prototype qui possède toutes les caractéristiques du langage, mais qui souffre encore de défauts de jeunesse particulièrement en ce qui concerne sa rapidité d'exécution et la place qu'il occupe en mémoire.

Notre but ici, n'est pas d'en décrire tous les détails, mais d'en donner quelques aperçus afin de répondre à des curiosités bien légitimes, et de donner une lueur plus technique sur la façon dont nous avons réalisé son implantation.

Le langage MERING a été implanté en VLISP sur PDP10 (Chailloux.78). De ce fait il utilise certaines des propriétés que l'on retrouve sur ce langage. Cependant il représente beaucoup plus qu'une simple extension de LISP (bien que dans notre implantation, toutes les fonctions LISP demeurent disponibles sans aucune altération). C'est un langage complet et autonome qui dispose de sa propre syntaxe, de ses structures de contrôles et de données.

IV.1.1 L'évaluateur

Toutes les entités MERING sont représentées sous la forme de fonctions LISP de la forme :

```
(df nom-de-l'entité (%msg%)
 (evaluer nom-de-l'entité))
```

où %msg% est la liste qui contient le message à évaluer, et où la fonction **evaluer** constitue l'évaluateur interne qui se charge d'interpréter le message, une technique déjà employée pour réaliser l'interpréteur OBJVLISP (Cointe.83). Grâce à

cette construction il est possible de conserver tous les avantages de LISP. En effet, toute transmission MERING revient à l'évaluation de la liste correspondante par la fonction **eval** standard, et donc bénéficie de la rapidité de l'interpréteur LISP sur lequel MERING est implanté.

Malheureusement si cette technique est envisageable pour toutes les entités explicitement décrites dans la base, il n'en est pas de même des entités de base telles que les nombres et les séquences. Par exemple, une transmission du type:

```
(3 + 2)
```

bien qu'évidente, ne peut être résolue avec cette méthode. De ce fait nous avons été amené à construire un réévaluateur, composés d'une fonction évaluatrice (**meval**) d'une boucle extérieure (**topmrg**) de la forme classique:

```
(de topmrg ()
  (print (meval (read)))
  (topmrg))
```

Les fonctions **meval** et **evaluer**, aidée de **mapply** constituent donc le coeur de l'interpréteur. La première, très semblable à la fonction **eval** de LISP, s'attache principalement à déterminer le type du receveur et à activer un évaluateur particulier dans le cas de nombres, de listes, de chaînes de caractères ou d'entité indéfinies, ou à passer la main à l'évaluateur standard de LISP dans tous les autres cas.

```
(defun meval (%form%)
  (cond
    ((or (numbp %form%)
         (stringp %form%)
         (litatom %form%))(eval %form%))
    (t (if (memq '& %form%) ; traitement du sequencement ;
            (mevprogn (generlst %form%))
            (if (memq '| %form%) ; traitement de la cascade ;
                (mevprogn (fmtcascade)))
            (let ((%obj% (next1 %form%)))
              (cond
                ((listp %obj%)(meval [(mevlist %obj%) . %form%]))
                ((numbp %obj%)(evalnum %obj% %form%))
                ((stringp %obj%)(evalstring %obj% %form%))
```

```
((memq (typefn %obj%)'
      '(subr fsubr array autoloader expr fexpr macro)
      (eval [%obj% . %form%] ))
  ((or (eq %obj% T)(eq %obj% nil)) (evalcond %obj% %form%
      (t (evalundef %obj% %form%)))))))))
```

Les fonctions **evaluer** et **mapply**, aidées de **getcomp** s'occupent du traitement d'une transmission dans le cas général. La première détermine s'il s'agit d'une A-transmission ou d'une M-transmission, récupère la définition du comportement déclenché par le message et demande à la fonction **mapply** d'effectuer la substitution des arguments et des symboles spéciaux

(% # ##)
par leur valeur réelles, et de "mévaluer" cette nouvelle expression.

```
(de evaluer (frame ;; sel typfn)
  (cond
    ; est ce un applicateur ? ;
    ((setq sel (getcomp frame (cadr %msg%) '$applic))
      (mapply frame (nextl sel) (nextl sel)
        (if (eq typfn 'fexpr)
          (cddr %msg%)(cddr %msg%))
        typfn
        (car %msg%)))
    ; est ce une methode ? ;
    ((setq sel (getcomp frame (car %msg%) '$methode))
      (mapply frame (nextl sel)(nextl sel)
        (if (eq typfn 'fexpr)
          (cdr %msg%)(cdr %msg%))
        typfn
        (car %msg%)))
    (t (send (print "** Erreur de transmission: [ frame . %msg%] )
```

```
(de mapply (frame fpar meth larg typfn slot)
  (setq meth (subst frame '% (subst slot '# meth)))
  ; traitement des comportements qui n'évaluent pas leurs arguments ;
  (if (eq typfn 'fexpr)
    (setq meth (subst ['quote larg] (car lpar) meth))
    ; traitement des comportements qui évaluent leur arguments ;
    (while lpar
      (if larg
        (cond ((listp (car larg))
              (setq meth (subst (nextl larg)
                                (nextl lpar) meth )))
              (t (setq meth (subst (nextl larg)
                                    (nextl lpar) meth))))))
        (meval meth))))))
```

Ces substitutions sont effectuées de manière physique sur le corps du comportement à évaluer, malgré son inefficacité dans un contexte d'application en grandeur réelle. En effet, nous avons signalé qu'il ne s'agit que d'un prototype pour lequel l'efficacité n'est pas un facteur essentiel. Cependant, nous projetons de réaliser une version efficace, et d'utiliser alors une technique d'évaluation dans un environnement à l'image de la plupart des langages d'acteurs.

IV.1.2 La structure des entités

Nous n'avons pas réalisé les attributs selon le modèle développé au chapitre II.4.4.1. En effet ce modèle a pris jour à partir d'une compréhension théorique d'un système fonctionnant déjà. Bien que nous planifions de réaliser un interpréteur selon ces principes, il nous est apparu plus important d'élaborer dans un premier temps un langage fonctionnant réellement, et capable de montrer sa puissance dans le domaine de la représentation des connaissances, plutôt que de mettre en oeuvre un modèle plus pur sur le plan théorique mais sans aboutissement immédiat.

De plus le modèle présenté au chapitre II.4.4.2 soulève de nombreuses difficultés de mise en oeuvre: il suppose un double mécanisme d'héritage (et cela n'a rien à voir avec l'héritage multiple), sur les attributs: le premier correspond à l'héritage normal sur les facettes (le contenu d'une facette d'un type ou d'un sur-type est directement hérité par ses représentants ou ses sous-types), le second aux définition par défaut des propriétés du type attribut, lesquelles sont partagées par tous les attributs.

De ce fait notre réalisation physique s'écarte du modèle théorique. Cependant cet écart n'implique pas de véritable distorsion entre la théorie et la pratique. Vu de l'utilisateur, tout se passe comme si le modèle théorique avait bien été implanté tel quel. En particulier les attributs disposent tous de facettes définies par défaut, qui peuvent être redéfinies pour une application particulière. Il suffit alors de redéfinir l'attribut \$slotini sur lequel sont chaînées toutes les recherches dans l'arborescence.

Les attributs et les comportements MERING, sont construits sur le modèle d'une a-liste de a-liste, c'est à dire que les informations associées à la définition d'une entité, qu'il s'agisse d'un type ou de l'un de ses représentants, sont mises sous la forme:

$$\begin{aligned}
 & ((\text{nom}_1 \quad (x_{1.1} \quad v_{1.1}) \\
 & \quad (x_{1.2} \quad v_{1.2}) \\
 & \quad \cdot \\
 & \quad (x_{1.n} \quad v_{1.n})) \\
 & (\text{nom}_2 \quad (x_{2.1} \quad v_{2.1}) \\
 & \quad \cdot \\
 & \quad (x_{2.m} \quad v_{2.m})) \\
 & \cdot \\
 & \cdot \\
 & (\text{nom}_k \quad (x_{k.1} \quad v_{k.1}) \\
 & \quad \cdot \\
 & \quad (x_{k.p} \quad v_{k.p})))
 \end{aligned}$$

où les 'nom_i' désignent aussi bien des noms d'attributs

que des sélecteurs de comportements.

Cette a-liste est ensuite placée sur la liste de l'atome correspondant à l'entité sous l'étiquette **frame**. Par conséquent une entité est définie à la fois comme une fonction, qui a pour objet l'évaluation de la liste qui lui est passée en argument (et sera évaluée selon la syntaxe MERING), et d'autre part comme une a-liste qui contient toutes les informations (attributs et comportements) nécessaires à sa définition.

L'utilisation d'une a-liste placée sur la p-liste est très pratique pour créer des environnements différents. Il suffit d'utiliser les fonctions ADDPROP et REMPROP pour créer de nouvelles entités masquant les anciennes ou revenir à une situation antérieure. C'est d'ailleurs comme cela que fonctionne la fonctionnelle MLET:

```
(dm mlet (ls)
  (rplacb ls
    ['progn
      ['addobj ['quote (mapcar (cadr ls) 'car)]
        ['quote (mapcar (cadr ls) 'cadr)]]
      ['progl
        ['mprogn . (cddr ls)]
        ['remobj ['quote (mapcar (cadr ls) 'car)]]]]))

(de addobj (lobj lclass)
  (while lobj
    (addprop (car lobj)
      ['lambda ['%msg%
        ['evaluer ['quote (car lobj)]]] 'fexpr)
    (addprop (car lobj)
      [(car lobj)
        '($type ( <- $inst))
        ['$isa [' <- (cond ((car lclass))
          (t '$objet))]]] 'frame)
    (nextl lobj)
  (nextl lclass)))
```

```
(de remobj (lobj)
  (while lobj
    (remprop (car lobj) 'expr)
    (remprop (nextl lobj) 'frame))) .
```

De ce fait, toutes les entités sont globales dans la base, et sont accessibles à tout moment. Cette conception qui se situe à l'inverse de la programmation structurée et qui peut déplaire à certains, est pourtant fort appréciée en Intelligence Artificielle où les raisonnements ne s'accordent pas toujours de visions "puristes". D'ailleurs, de nombreux chercheurs pensent que l'avenir n'est pas à une programmation rigide et par trop structurée, mais à une programmation "chevelue", où les modes de communications tiennent plus du dialogue de personnes en "sociétés" que d'un mode de contrôle formel et très typé dans le style ADA.

Les informations associées aux facettes sont directement décrites dans les couples (x_i, y_i) . Il en est de même pour les sélecteurs: les couples définis par les x_i , \$methode et \$applic contiennent la définition des méthodes et des applicateurs respectivement, et la facette \$type indique s'il s'agit d'un comportement qui évalue ses arguments ou non.

la structure interne d'un comportement (par exemple une méthode) est la suivante:

```
(nom-comportement ($methode (args) corps-du-comportement)
  ($type type-du-comportement))
```

la recherche des attributs ou des comportements dans l'arborescence est effectuée lors de l'interprétation. Cette technique est plus coûteuse en temps, mais autorise une souplesse accrue: toutes les techniques qui tendent à compiler d'une manière ou d'une autre la liste des types dont hérite une entité, tend à rigidifier le système. C'est d'ailleurs la principale critique qui pourrait être portée contre des langages comme GLISP, qui cherchent à compiler au maximum toutes les informations dont ils disposent. Evidemment, ce qu'ils perdent en souplesse, ils le gagnent en rapidité.

Cette recherche des sélecteurs est réalisée sous la forme d'un parcours en profondeur d'abord, ce qui permet d'attribuer une priorité aux types pères situés au début de la liste, et donc de limiter les risques de conflits lors de la présence d'attributs (ou de comportements) de noms semblables dans le treillis.

La fonction de base de recherche d'une information, ramène une valeur associée à une facette, connaissant l'entité, le sélecteur et la facette. Cette fonction, très classique dans les systèmes de schémas s'intitule **fget-i**. Son rôle consiste à appeler la fonction **fgetclasses**, qui fournit la liste de tous les types situés au dessus de l'entité en question, et à examiner cette liste:

```
(de fget-i (frame slot facet)
  (let ((classes (fgetclasses frame))(result))
    (while classes
      (if (setq result (fget (car classes) slot facet))
        (lescape result)
        (nextl classes) ))
      (fget '$sysobj '$slotini facet) ; chainage du slotini;
    ))))))))
```

Le chainage avec l'attribut par défaut (slotini), est réalisé à la fin de cette exploration. La fonction **fget** sert à récupérer une information à un niveau précis, sans tenir compte des mécanismes d'héritages.

Cependant cette recherche peut être modifiée par l'emploi de la fonction **super:**, qui modifie le point de départ des recherches dans l'arborescence des ascendants. Cette fonction qui prend pour argument un sélecteur et un nom d'entité, ramène ce nom s'il est différent de nil, ou bien le nom du type situé au dessus du type de référence (c'est à dire le type du type, si l'entité en question est un représentant, ou le type immédiatement supérieur dans le cas contraire).

La fonction de base d'affectation d'une valeur à une facette est la fonction **fset**. Elle effectue une modification physique de la a-liste contenant les informations de l'entité:


```
(de fset (entite attribut facette1 valeur)
  (rplacd (fassoc (facette attribut (fget 'entite 'frame)))
    (if (atom valeur) [valeur] valeur))
  entite)
```

La fonction **fassoc** est une fonction utilitaire qui place physiquement une cle sur une a-liste si celle ci n'est pas encore présente sur cette liste:

```
(de fassoc (cle aliste)
  (cond ((assoc cle (cdr aliste)))
    (t (cadr (rplacd (last aliste) [[cle]]))))))
```

IV.1.3 Boucle externe et continuations

Les continuations ont été réalisées en utilisant la structure des échappements déjà présents en VLISP. La fonction **send**: a seulement pour effet de placer le message qui lui est envoyé dans une variable globale **%message%** qui contient la transmission à évaluer, puis elle effectue une sortie de l'échappement qui est déclaré au niveau de la boucle externe (c'est à dire par la fonction **topmrg**, le 'toplevel' de MERING), en signifiant à cette boucle qu'elle doit évaluer le message placé dans **%message%**.

La fonction **topmrg** est construite autour d'une double boucle. La plus externe correspond à la fonction **toplevel** classique dans le style (print (eval (read))), tandis que la boucle interne se charge d'évaluer les continuations placées dans **%message%**.

```
(df send: (%m%)
  (setq %message% (cdr %m%))
  ($return '$cont))
```

```
(de topmrg ( ;; S)
  (while T
    (setq s (read))
    (while (eq (setq s (escape $return (meval s))) '$cont)
      (setq s %message%))
    (print s)
    (nettoie))))
```

Cette double structure permet donc de programmer selon deux s: dans le premier, les évaluations sont exécutées alement en empilant chaque appel à un comportement. Au raire dans le second, l'appel de la fonction **sendM** a résultat de dépiler tout l'environnement pour revenir au au supérieur et d'évaluer les transmissions au niveau de la le extérieure, comme des messages envoyés entre acteurs.

Grâce à une telle structure de contrôle, il serait le de créer un mécanisme d'agenda (Charniak.80) en idérant la variable %message% comme une liste de missions en attente d'être exécutées, ou mieux encore e une entité MERING destinée à contrôler les échanges de ages entre acteurs. Une telle technique permettrait planter aisément un système de co-routines.

La fonction **nettoie** s'occupe de récupérer la place occupée les entités fantomes, principalement les listes receveuses, explicitement décrites. En effet, l'évaluation d'une mission, par exemple

```
('(a b c d) l?)
```

conduit à la génération d'une séquence contenant les objets o, c et d, mais dont la durée de vie éphémère, est réduite à e seule transmission. Cette entité doit être ensuite uite afin de ne pas encombrer l'espace de travail. C'est ce oyage", effectué après chaque retour au niveau de la boucle ieure que décrit la fonction **nettoie**. Bien que arence éphémère, il est possible de montrer la présence de entités fantomes:

```
; ('(a b c d) print | l?)
($SEQ => $SEQ1
  CELL <- (A B C D)
)
```

A

```
:$seq1 print
ENTITE INDEFINIE: $SEQ1
```

Les entités de type **\$seq** et **\$cell** sont définies comme éphémères, et comme telles sont éliminées à chaque tour par la fonction de nettoyage.

IV. 1.4 Les fonctionnelles

Les fonctionnelles de contrôle (if, while, mprogn, mevprogn, etc..) sont écrites à l'aide de macros ou de FEXPR (c'est à dire des fonctions LISP qui n'évaluent pas leurs arguments) qui reprennent directement les codes classiques pour de telles fonctions.

Les fonctionnelles FORALL, EXIST et TESTALL sont réalisées grâce à des fonctions de type FEXPR qui effectuent des substitutions dans le corps de la fonctionnelle:

```
(df forall (%1%)
  (let ((s (meval (cadr %1%))))
    (if (atom s)
        (mevprogn (subst s (car %1%) (cddr %1%) ))
        (mapct s
          '(lambda (%X%)
            (mevprogn (subst %X% (car %1%) (cddr %1%)))))))

(de exist (%1%)
  (let ((s (meval (cadr %1%))))
    (if (atom s)
        (mevprogn (subst s (car %1%) (cddr %1%)))
        (some s
          '(lambda (%X%)
            (mevprogn (subst %X% (car %1%) (cddr %1%)))))))
```

```
(de testall (%l%)
  (let ((s (meval (cadr %l%))))
    (if (atom s)
        (if (mevprogn (subst s (car %l%)(cddr %l%))) T)
        (if (every s
                    '(lambda (%X%)
                        (mevprogn (subst %X% (car %l%)(cddr %l%)))))
            T))))))
```


IV.2 Manuel de référence

MERING lors de son chargement contient plusieurs types pré-définis dont `$sysroot` qui est la racine du langage et le père de tous les autres types,

type, qui est le méta-type du langage et dont les représentants sont eux-mêmes des types, et un certain nombre d'autres types qui constituent l'environnement de programmation de MERING.

Ces types d'origine contiennent tous les comportements de base (méthodes et applicateurs) nécessaires au bon fonctionnement du langage.

La figure 4.1 présente la hiérarchie existante entre les différents types de base. Certains comme **regle** et **classe** ne sont pas traités ici, mais sont examinés dans le chapitre III, car il faut à proprement parler d'extensions du langage.

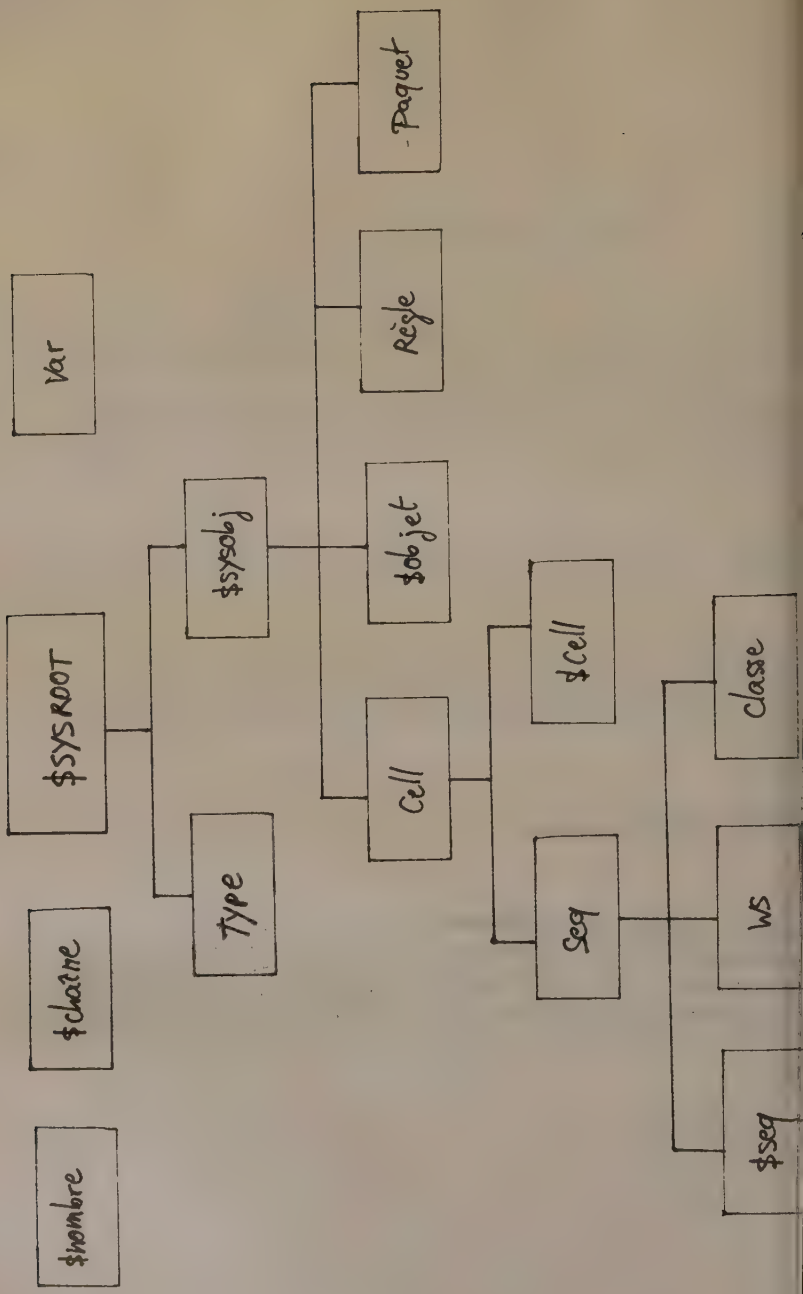
IV.2.1 Les entités de base

Les **nombre**s et les **chaines** de caractères constituent deux entités de base du système. Les méthodes qu'elles connaissent sont assez classiques: opérations arithmétiques et comparaisons pour les nombres, concaténation, suppression de caractères, recherche ou obtention d'une sous-chaine, pour les chaines de caractères.

Ex:

```
: (2 + 3)
= 5
```

```
: (2 * (3 + 4))
= 14
```



```
: ((10 / 2) / 2)
= 2.5
```

```
: ("bon" * "jour")
= "bonjour"
```

- **\$sysroot** est l'entité fondamentale du système, celle dont toutes les entités ultérieures sont issues (à part les nombres, les chaînes de caractères et le type indéfini). Celle ci comporte les méthodes et les applicateurs les plus généraux. Affectation, ajout, lecture et suppression de la, ou des, valeur d'un attribut, identification du type et de la catégorie de l'entité, sa représentation externe, impression du comportement traitant un message particulier, listes des comportements, méthodes, applicateurs et des attributs que connaît une entité, etc... Ce type est en généralement invisible pour l'utilisateur.

- **\$sysobj** est en fait le type de base perçu par un utilisateur. Presque tous les types en sont issu. Il comporte l'ensemble des applicateurs et des méthodes permettant une manipulation efficace de la base. Ce type est entièrement manipulable par l'utilisateur. En particulier l'attribut auxquels sont chaînés tous les autres et qui contient les valeurs par défaut des facettes (**\$slotin**) se trouve dans ce type. L'utilisateur, pour une application particulière pourra être amené à redéfinir cet attribut (il est d'ailleurs donné dans ce but). Sa description par défaut est la suivante:

```
$slotini
$dom      $indef
$default  nil
$quant    nil
$req      nil
$valeurs  nil
$mode     $mult
$ordre-lect ($default $sireq $regle-req -)
$si-req   nil
$si-lu    nil
$si-ajout nil
$si-suppr nil
$regle-req nil
$regle-ajout nil
```



```
$si-err-ajout  (** erreur ajout valeur dans : " prnl
               & % prnl & # prnl & terpri & send: %)
$si-err-mode   (** erreur ajout valeur univaluee dans : " pri
               & % prnl & # prnl & terpri & send: %)
$question     (quelle est la valeur du # de %),
```

- **\$objet** est le type par défaut de création d'un nouveau type. Ce type ne contient quasiment aucun comportement. Il est utilisé comme relais entre les entités systèmes et les entités utiles à une application.

- **type** est le méta-type dont tous les types sont les représentants. Il n'est utilisé principalement que pour créer de nouveaux types. Il sert aussi à comptabiliser des éléments statistiques concernant la création et la suppression d'entités. Ce type ne connaît aucun sous-type.

- **var** est un type de base qui n'est issu d'aucun autre, et qui n'a aucun sous-types. Ces représentants ne sont pas des entités à proprement parler, mais des références à des entités, des variables globales qui se comportent comme les entités qu'elles contiennent. Elle ne répond en tant que telle qu'à très peu de messages. Cependant dès qu'une entité lui est affectée, elle répond directement aux messages que connaît cette entité, et agit alors de manière totalement transparente.

Le seul message que connaît ce type est => qui sert à la création d'un nouveau représentant, une nouvelle variable. Ces dernières répondent en propre aux messages suivant:

:= est l'affectation d'une entité à une variable. Par exemple

```
(a := employe)
```

attribue l'entité 'employe' à la variable 'a'. Toute référence à cette référence sera interprétée comme s'il s'agissait du type employe lui-même, jusqu'à son affectation par une autre entité. En particulier le message

```
(a => alfred)
```

est alors absolument équivalent à

```
(employé => alfred)
```

pvar est une méthode qui ramène T lorsque l'entité est une variable, et NIL dans tous les autres cas. Il s'agit d'un test pour savoir s'il s'agit d'une entité authentique ou d'une référence à cette entité.

```
: (a pvar)
= T
: (employé pvar)
= NIL
```

Tous les autres messages sont dérivés vers l'entité que la variable en question supporte.

cell est le type dont les représentants servent à contenir des données, à l'image de la cellule PLASMA. Presque tous les applicateurs décrits dans **\$sysroot** et **\$sysobj** se trouvent ici sous la forme de méthodes. En réalité un attribut implicite dans ces messages, il s'agit de **cell** qui contient les informations de la cellule. L'affectation, la lecture et l'ajout sont ici des méthodes.Ex:

```
: (cell => a | <- 4)
= A
: a
= 4
: (a ?)
= 4
: (a is)
= CELL
: (a print)
  (CELL => A
    $CELL :: (4),
  )
= A
```

ne faut pas confondre les **cell** et les **var**, car ils ne

réagissent pas du tout de la même manière aux messages qui leur sont adressés. Un message envoyé à une **cell** est interprété comme étant véritablement envoyé à cette entité, alors qu'un message adressé à une entité de type **var** est dérouter vers l'entité qu'elle référence:

```
: (var => | := (personne => toto))
= TOTO

: (cell => b | <- (personne => riri))
= B

: (a print)
  (PERSONNE => TOTO
  )
= TOTO

: (b print)
  (CELL => B
    CELL <- (RIRI),
  )
= B
```

- **seq** introduit le type de donnée: la séquence. Celle ci a les mêmes caractéristiques que la liste Lisp. Elle permet en plus d'être manipulée comme un vecteur c'est à dire un tableau uni-dimensionnel. Ex:

```
: (seq => toto | <- '(a b c d) | print)
  (SEQ => TOTO
    CELL <- (A B C D),
  )
= TOTO

: (toto <= e | ?)
= (A B C D E)

: (toto !!) ; identique au NEXTL Lisp ;
= A

: (toto ?)
= (B C D E)

: (toto count)
= 4
```

- **\$cell** et **\$seq** sont les type qui désigne les cellules et séquences temporaires (ou fantomes) qui sont régulièrement détruites lors du retour au "toplevel".

- **ws** est un sous-type de **seq** qui gère le chargement et la sauvegarde des entités sous la forme d'espaces de travail, un peu à la manière des 'workspace' du langage APL. Les deux méthodes fondamentales de ces entités sont **load** et **save** qui charge un fichier et le place dans un espace de travail, ou sauvegarde un ensemble d'entités sur un fichier texte.

Ex:

```
: (ws => machin | load "machin.vli")
= "machin.vli"
: (machin save "truc.vli")
= "truc.vli"
```

- **undef** est le type qui regroupe toutes les entités qui n'ont pas été définies comme tel. Lorsqu'un message est envoyé à un symbole qui n'est pas une entité, ni un nombre, ni une chaîne de caractères, Mering suppose qu'il s'agit d'une entité non encore définie. Celle ci réagit à certains messages qui permettent de lui attribuer un statut.

:= estime que l'entité indéfinie est en fait une variable, auquel elle attribue comme valeur l'entité qui suit.

<- ou **<=** la définit comme une séquence ou cellule suivant que la valeur de l'argument du message est une liste ou un simple atome (éventuellement une entité). Cet argument est alors placé dans l'entité nouvellement créée.Ex:

```
: (a kill)
= A
: (a <- 'b | is)
= CELL
: (a kill)
= A
: (a <- '(b c d) | is)
= SEQ
```

? retourne NIL, et is ramène 'undef'

A ces entités de base, il faut rajouter toutes les extensions déjà programmées (classes, règles de production et paquets de règles, décrites dans le chapitre III, les filtres, entités servant au filtrage (pattern matching) qui ne sont pas présentées ici), et celles que nous projetons d'ajouter au système: structures de contrôles par agendas, co-routines, superviseur pour suivre le déroulement des transmissions, etc...

IV.2.2 Description des comportements de base

MERING comprend un certain nombre de comportements (méthodes et applicateurs) de base, qui font partie du langage lui-même. Ils se trouvent dans les types `$sysroot` et `$sysobj`, et sont donc utilisables par toutes les entités de la base. D'autres sont plus spécifiques à des types représentant des structures de données tels que les types `cell` ou `seq`

Nous donnons ici la liste des comportements les plus important en notant pour chacun d'eux, le nom, le nombre d'arguments, le type de comportement (**methode:** et **applic:** pour les comportements qui évaluent leurs arguments, et **methode|** et **applic|** pour ceux qui n'évaluent pas leurs arguments), ainsi que leurs caractéristiques fonctionnelles:

Ces comportements se rangent en plusieurs catégories:

- methodes générales
- Tests de types
- Manipulation de facettes
- Applicateurs actifs

- Prédicats
- Manipulation de listes (générales, indexées et associatives)

Les méthodes générales

Cette catégorie comprend toutes les méthodes de base qui s'appliquent à toutes les entités: création de nouvelles entités, visualisation de la structure d'une entité, méthodes déclenchées lors de la génération d'une entité, etc...

=> **methode** | (**lst**) : crée un représentant (ou un type si le receveur est le méta-type 'type'). La définition de l'entité créée se trouve dans la liste de tokens 'lst', qui n'est pas évaluée.

L'application de cette méthode déclenche la méthode 'newtype' s'il s'agit de la création d'un type, et 'newrep' dans la cas d'un représentant. Ces méthodes peuvent être réécrites par l'utilisateur de manière à initialiser des attributs par exemple. Le résultat normal de ces méthodes est de ramener l'entité créée. Ex:

```
: (type => animal elephant
  partie <= trompe,
  regime <- herbivore,
  reproduction $dom mode-reprod
                <- mammifere,
)
= ELEPHANT

: (elephant => jumbo
  newrep methode: () ("salut les copains" print & %),
)
"salut les copains"
= JUMBO
```

kill methode: () : supprime l'entité receveuse de la base.
 Elle prend alors le type '\$indef'. Attention, l'utilisation de cette méthode de manière inconsidérée est dangereuse. En effet elle n'élimine pas toutes les références qui ont trait à cette entité et sont disséminées dans toute la base. D'autre part, si l'entité en question est un type, tous ses sous-types, ainsi que les représentants qui s'y rapportent risquent de devenir isolés des autres éléments, et d'être incapable de recevoir et d'interpréter les messages qu'elles connaissent par héritage. Il s'agit donc d'une primitive de base qui peut être utilisée pour construire des comportements d'effacement plus sophistiqués.

print methode: () : livre le contenu de l'entité sous une forme lisible dont la syntaxe est identique à celle de sa définition. De ce fait, une entité peut être sauvée sur fichier et relue par la suite, sans nécessiter de modifications. Cette méthode retourne l'entité décrite comme résultat. Ex:

```

: (elephant print)
  (TYPE => ANIMAL ELEPHANT
    PARTIE :: (TETE TRONC MEMBRES TROMPE),
    REGIME :: (HERBIVORE),
    REPRODUCTION $DOM MODE-REPROD
                :: (MAMMIFERE),
  )
= ELEPHANT
    
```

:= methode: (E) est une méthode dangereuse. Elle a pour fonction de définir une entité comme 'variable' avec E (qui doit être une entité) comme contenu. Elle se décrit ainsi:

```

:= methode: (E) (var => % | := E),
    
```

listattr methode: () : liste tous les attributs non système (c'est à dire qui ne sont pas précédés du caractère '\$') de l'entité receveuse.

```
: (jumbo listattr)
= (PARTIE REPRODUCTION REGIME)
```

Listmeth methode: () et **listappl methode: ()** sont deux méthodes qui ont pour fonction respectives de lister l'ensemble des méthodes et des applicateurs que connaît l'entité en question.

all methode: () est une méthode qui retourne l'ensemble de ses représentants directs, si l'entité est un type, ou bien l'ensemble des représentants frères (c'est à dire issus du même type que l'entité en question) si le receveur du message est un représentant.

```
: (elephant => riton)
= RITON
```

```
: (animal all)
= NIL
```

```
: (elephant all)
= (JUMBO RITON)
```

```
: (jumbo all)
= RITON
```

allinst methode: () Si l'entité receveuse est un type, cette méthode retourne l'ensemble de ses représentants directs et indirects (c'est à dire issus de sous-types). Autrement elle ramène simplement cette entité.

```
: (animal allinst)
= (JUMBO RITON)
```

```
: (jumbo allinst)
= JUMBO
```

allent methode: () retourne l'ensemble des entités (types et représentants) situées hiérarchiquement en dessous, y compris l'entité receveuse elle même. Les représentants

sous placés en premier, les types ensuite.

```
: (animal allent)
= (JUMBO RITON ELEPHANT GAZELLE ANIMAL)

: (jumbo allent)
= JUMBO
```

firstinst methode: () ramène le premier représentant direct (c'est à dire le premier créé) du receveur lui-même si le receveur est un type, ou du type auquel appartient le receveur s'il s'agit d'un représentant.

```
: (elephant firstinst)
= JUMBO

: (animal firstinst)
= NIL ;ne marche que pour le représentant direct ;
```

lastinst methode: () ramène le dernier représentant créé dans les mêmes conditions que la méthode précédente.

```
: (elephant lastinst)
= RITON
```

hierarch methode: () imprime l'ensemble du treillis hiérarchique en indiquant pour chaque entité, s'il s'agit d'un représentant ou d'un type, l'ensemble des types dont elle est issue, et, s'il s'agit d'un type, l'ensemble de ses sous-types et de ses représentants.

newrep methode: () est un comportement déclenché automatiquement à la création d'un représentant. Cette méthode peut être réécrit par l'utilisateur pour s'adapter à ses besoins, initialisation de paramètres par exemple. De manière générale elle ne fait que ramener le nom de l'entité nouvellement engendrée. Elle est décrite ainsi:

```
newrep methode: () ('%),
```

newtype methode: () est identique à 'newrep' si ce n'est qu'elle est déclenchée lors de la création d'un type et non d'un représentant.

questionner applic: () permet de réaliser simplement des dialogues avec l'utilisateur. Ce comportement affiche d'abord le contenu de la facette \$question, puis attend une réponse. L'information entrée par l'utilisateur est ensuite placée dans l'attribut en question.

IV.2.2.2 Tests de types

Ces comportements servent à tester le type dont est issu l'entité en question. En particulier savoir si c'est un nombre, une chaîne, etc... Ils concernent aussi la catégorie, le méta-type, auquel appartient cette entité: type ou représentant.

Tous ces comportements existent sous les deux formes, méthodes ou applicateurs. Lorsqu'il s'agit de la forme applicative, le test en question n'est plus effectué sur l'entité receveuse, mais sur la valeur ou le domaine de l'attribut référencé par la transmission.

is () : retourne le type dont est issu l'entité concernée. Ex

```
: (elephant => jumbo)
= JUMBO
```

```
: (jumbo is)
= ELEPHANT
```

```
: (elephant is)
= ANIMAL
```

```
: (toto age is)
= $NOMBRE
```

(v) : indique si l'entité en question est issue directement ou indirectement du type 'v'.

```
: (jumbo isa? animal)
= T
```

```
: (jumbo isa? plante)
= NIL
```

```
: (jumbo reproduction isa? mode-reprod)
= T
```

rep? methode: () : indique si l'entité ou la valeur de l'attribut est un représentant. Si l'attribut est vide (valeur NIL) ou s'il contient, parmi plusieurs valeurs, un type, la valeur retournée est NIL.

```
: (jumbo rep?)
= T
```

```
: (elephant rep?)
= NIL
```

```
: (elephant reproduction rep?)
= T
```

```
: (jumbo partie rep?)
= NIL
```

type? () : indique si l'entité, ou la valeur de l'attribut, est un type ou non. Si l'attribut contient parmi ses valeurs, représentant, la valeur retournée est NIL. En revanche, l'attribut est vide, la valeur est T.

```
: (elephant type?)
= T
```

```
: (jumbo type?)
= NIL
```

```
: (animal reproduction type?)
= T
```

```
: (jumbo reproduction type?)
= NIL
```

nombre? methode: () : indique si l'entité est un nombre, ou s'il s'agit d'une **cell** qui contient un nombre.

```
: (3 nombre?)
= T
: (cell => a | <- 4 | nombre?)
= T
```

nombre? applic: () : teste si l'attribut receveur contient un nombre, ou une **cell** qui contient un nombre. De plus, si son domaine de définition est **\$nombre**, la réponse est affirmative même si l'attribut ne contient aucune valeur..

```
: (toto age nombre?)
= T

: (jumbo partie nombre?)
= NIL

: (toto truc <- (cell => | <- 432) | truc nombre?)
= T
```

chaîne? () : identique au comportement précédent, mais teste ici s'il s'agit d'une chaîne de caractères.

indef? methode: () : teste si l'entité est de type indéfini.

```
: (a kill)
= A

: (a indef?)
= T

: (jumbo indef?)
= NIL
```

indef? applic: () retourne vrai (T) si:

1) L'attribut est vide et son domaine est '\$indef' ou NIL ou '\$valeurs'.

2) l'attribut contient une ou plusieurs valeurs n'étant pas reconnue comme des entités de la base (c'est à dire de simples atomes non numériques ou des listes).

```
: (animal reproduction indef?)
= NIL
```

```
: (animal regime indef?)
= T
```

```
: (toto poids indef?)
= T
```

```
: (toto poids <- 78 | poids indef?)
= NIL
```

entite? () est définit comme l'inverse de 'indef?'.

```
entite? methode: () (not (% indef?)),
entite? applic: () (not (% # indef?)),
```

var? methode: () teste si l'entité receveuse est une variable qui contient une entité.

```
: (jumbo var?)
= NIL
```

```
: (a := jumbo)
= JUMBO
```

```
: (a var?)
= T
```

IV.2.2.3 La manipulation des facettes

Les opérations d'affectation ou de lectures d'informations associées à une facette particulière d'un attribut, sont régies par des applicateurs dont la sémantique est fort simple car ils ne déclenchent aucun réflexe.

Le premier de ces applicateurs, **nulfac?** est un test appliqué à une facette pour savoir si sa valeur est nulle, c'est à dire qu'elle ne contient aucune information:

```
: (personne => toto
  age <- 40,
)
= TOTO

: (toto age nulfac? '$default)
= T

: (toto age nulfac? '<-)
= NIL
```

L'affectation est réalisée grâce à l'applicateur **<-fac:** qui place une donnée dans la facette dont le nom est passé en argument.

```
: (toto age <-fac: '$default 30)
```

est une opération qui associe la valeur par défaut 30 à l'attribut 'age' de 'toto'.

```
: (toto print)
(PERSONNE => TOTO
  AGE    $DOM      ($NOMBRE)
        $DEFAULT  (30)
        <-       (40),
)
= TOTO
```

La lecture de données, opération réciproque de la précédente s'effectue grâce à l'applicateur **fac?**

```
: (toto age fac? '$default)
= 30
```

Cette manipulation agit donc au niveau de la facette et non à celui de l'attribut. De plus elle utilise les relations d'héritage qui instituent une véritable "copie virtuelle" des informations placées dans les entités situées au dessus dans la hiérarchie. Par exemple, si nous reprenons nos entités animalières :

```
: (animal regime <-fac: '$default carnivore)
= ANIMAL

: (jumbo regime ?)
= HERBIVORE

: (jumbo regime fac? '$default)
= CARNIVORE
```

Il est cependant possible de masquer cet héritage, en employant l'applicateur **fac??** à la place de **fac?**. Celui ci ignore les mécanismes de copie virtuelle et s'adresse directement à l'entité receveuse du message.

```
: (jumbo regime fac??)
= NIL
```

L'information n'ayant pas été fournie à ce niveau de la hiérarchie, la valeur retournée est NIL.

De plus l'emploi de cet applicateur supprime toutes les valeurs par défaut que possèdent toutes les facettes des attributs et qui sont déclarées dans **\$slotini** contenu dans le type **\$sysobj**. Rappelons que cet attribut est automatiquement chaîné à tous les autres lors de la recherche dans l'arborescence, et qu'il introduit ainsi un mécanisme d'informations par défaut valable pour tous les attributs de toutes les entités.

```
: ($sysobj $slotini <-fac: '$default rien)
= $SYSOBJ

: (jumbo reproduction fac? '$default)
```

= RIEN

```
: (jumbo reproduction fac?? '$default)
= NIL
```

A ce niveau, les facettes se comportent de manière anonyme: qu'il s'agisse de facettes valeurs, filtres ou réflexes, le système n'en a cure. Il considère que les informations qui sont placées et lues dans ces facettes, ne sont que des s-expressions.

De ce fait, n'importe quoi peut être placé dans n'importe quelle facette, le système l'acceptera. En effet, ces applicateurs sont conçus comme des opérateurs primitifs à partir desquels se bâtissent des comportements plus sophistiqués. Nous examinerons cette élaboration dans la partie suivante concernant les applicateurs actifs.

Deux autres applicateurs primitifs sont connus du système: **<=fac:** et **facdel**. Le premier ajoute des données à une facette alors que le second en supprime.

```
: (animal regime <=fac: '$default insectivore)
= ANIMAL
```

```
: (jumbo regime fac? '$default)
= (CARNIVORE INSECTIVORE)
```

```
: (animal regime facdel '$default carnivore)
= CARNIVORE
```

```
: (jumbo regime fac? '$default)
= INSECTIVORE
```

Les applicateurs **<-fac:** et **<=fac:** existent aussi sous une version qui n'évalue pas ses arguments: **<-fac|** et **<=fac|**. Ces versions sont très utiles pour placer des transmissions dans des facettes réflexes au cours du déroulement du programme.

```
: (jumbo age <-fac| $sireq (toto age ?))
= JUMBO
: (jumbo age fac? $sireq)
```



```
= (TOTO AGE ?)
: (jumbo age ?)
= 40
```

Dans cet exemple, l'age de 'jumbo' est considéré comme étant identique à celui de 'toto'.

On retrouve souvent le besoin de placer ou de lire des informations dans les mêmes facettes. C'est pourquoi un certain nombre d'applicateurs ont été créés pour effectuer directement ces opérations sans devoir passer le nom de la facette en argument. Ces comportements ont été réalisés surtout pour les facettes \$default, \$dom, \$si-req et \$si-ajout. Bien sûr, cette liste n'est pas limitative et pourrait être facilement allongée si le besoin s'en fait sentir. Voici leur définition:

```
<-def: applic: (v) (% # <-fac: '$default v),
<=def: applic: (v) (% # <=fac: '$default v),
def?? applic: () (% # fac?? '$default ),
defdel applic: (v) (% # facdel '$default ),
<-dom: applic: (v) (% # <-fac: '$dom v),
.
.
<-req: applic: (v) (% # <)fac: '$req v),
.
.
<-si-req applic| (v) (% # <-fac| $si-req v),
.
.
<-si-ajout applic| (v) (% # <-fac| $si-ajout v),
.
.
```

Ces applicateurs, par soucis d'efficacité n'ont pas été écrits à partir des manipulateurs de facettes, mais directement à partir des fonctions LISP qui ont servi à construire MERING.

IV.2.2.4 **Applicateurs actifs**

Ces comportements sont les éléments essentiels de la manipulation des attributs. Pour ce faire ils utilisent l'ensemble des informations disposées dans les facettes, filtrent les données qui sont associées à l'attribut, déclenchent des réflexes s'il y a lieu, etc...

Comme nous l'avons mentionné dans la section concernant le modèle théorique des attributs (chapitre II.4.4.1), les applicateurs doivent être compris comme des redistributeurs de messages provenant de l'extérieur vers des entités internes: les attributs. De ce fait ils jouent un rôle fondamental car ils décrivent les algorithmes de déclenchement des réflexes.

Tous ces applicateurs peuvent être réécrits pour satisfaire une application particulière. Par exemple, on pourra envisager un système d'affectation qui déclenche d'autres réflexes que ceux que nous mentionnons (pour contraindre les valeurs permises par d'autres systèmes de filtrage notamment). Pour cela il suffira de placer leur définition dans un type, et tous les sous-types et leurs représentants hériteront de ces nouvelles définitions. L'applicateur original, laissé intacte, sera toujours accessible normalement à partir d'autres entités non issues du type en question, ou bien dans ses descendants, en utilisant l'indicateur **super**:

Ces applicateurs sont décrits dans **\$sysroot**.

Ces applicateurs fondamentaux sont au nombre de cinq:

- 1) la lecture normale (?)
- 2) la lecture réduite (??) qui inhibe le réflexe associé à la facette \$si-lu.
- 3) l'affectation d'une valeur à l'attribut (<-)
- 4) l'ajout d'une valeur (<=)

5) la suppression d'une valeur (**del**)

IV.2.2.4.1 la lecture d'attributs

Le premier de ces applicateurs (?) est fondamental en Mering. Il a pour objet de retourner la valeur associée à l'attribut, c'est à dire en général l'information associée à sa facette '<-' (ou \$val). Mais son effet ne s'arrête pas là: c'est lui qui dirige l'activation de réflexes dont le but est de fournir des informations lorsque l'attribut est vide, c'est à dire lorsque la transmission

(entité attribut fac? '<-')

ramène la valeur NIL.

Dans ce cas il dispose de plusieurs facettes qui concourent à pallier cette carence d'informations: \$si-lu, \$default, \$si-req, \$regle-req et \$ordre-lect

Nous avons vu au chapitre II.4.4.3 la fonction de ces facettes. Nous allons maintenant examiner comment MERING les utilise lors de ses tâches de lecture.

Voici l'ordre des opérations de lecture:

- 1) si la facette \$si-lu est non-vide, alors évaluer son contenu et sortir
- 2) si la facette \$val (ou <-) est non-vide, alors ramener sa valeur et sortir
- 3) soit s le contenu de la facette \$ordre-lect et f le dernier élément de s
- 4) si s est vide alors aller en 9)
- 5) si premier(s) = \$default, alors si la facette \$default est non vide, ramener sa valeur et aller en 9), sinon

faire s = reste(s) et aller en 4)

- 6) si premier(s) = \$si-req, alors si la facette \$si-req est non vide, alors evaluer son contenu. Si le résultat est non nul alors ramener sa valeur et aller en 9), sinon faire s = reste(s) et aller en 4)
- 7) si premier(s) = \$regle-req, alors si la facette \$regle-req est non vide, alors exécuter chacune des règles qui s'y trouvent. Puis sortir
- 8) aller en 4)
- 9) si f = '+' alors si la facette \$regle-req est non vide alors exécuter chacune des règles qui s'y trouvent. Puis sortir

En d'autres termes on regarde d'abord si il ya une facette \$si-lu. Si elle existe son contenu est évalué, court circuitant ainsi le reste des autres opérations. L'emploi de cette facette se révèle très utile pour la réalisation d'héritages horizontaux, tel que celui qui correspond aux relations transitives (par ex: les parties des parties sont aussi des parties) comme nous l'avons décrit au chapitre II.4.4.2

Ensuite, en suivant l'ordre des facettes placées dans '\$ordre-lect', on ramène le contenu de la facette '\$default' ou la valeur retournée par le réflexe '\$si-req'. Puis on réalise l'activation des règles placées dans '\$regle-req' s'il y a lieu (c'est à dire si aucune valeur n'a été ramenée auparavant, ou si l'indicateur final est un signe '+').

L'applicateur ?? est très semblable à ?, si ce n'est qu'il n'effectue pas l'opération n°1, c'est à dire qu'il n'évalue pas le contenu de \$si-lu même s'il est non vide.

Soit le type suivant :

```

: (type => $objet bateau
  type $dom $indef
    $ordre-lect ($default $si-req $regle-req -)
    $default cargo
    $si-req (mselectq (ce fonction)
      (marchande 'cargo)
      (passagers 'paquebot)
      (guerre 'navire-guerre)),
  fonction $valeurs (marchande guerre passagers)
    $default (passagers),
  age-capit $dom $nombre
    $default 25
    $si-lu (toto age ?)
    $si-req ("salut" print),
)

```

L'ordre de lecture des facettes est gouverné par la facette **\$ordre-lect**:

```

: (bateau => ivre)
= IVRE

: (ivre type ?)
= CARGO

: (ivre fonction ?)
= PASSAGERS

```

C'est en effet la facette **\$default** qui se présente la première dans les attributs **type** et **fonction**. Maintenant si l'ordre de cet attribut est modifié c'est la facette **\$si-req** qui sera utilisée en premier.

```

: (ivre type <-fac: '$ordre-lect '($si-req $default))
= IVRE

: (ivre type ?)
= PAQUEBOT

```

La facette **\$si-lu** court circuïte l'emploi de toutes les autres

```

: (toto age <- 40 & ivre age-capit ?)
= 40

```

En revanche, l'utilisation de l'applicateur ?? ignore le contenu de cette facette.

```
: (ivre age-capit ?)
25
```

Modifier l'ordre de lecture agit aussi sur cet applicateur.

```
: (ivre age-capit <-fac: '$ordre-lect '($si-req $default - ))
= IVRE

: (ivre age-capit ??)
= "salut"
```

IV.2.2.4.2 Affectation et ajout de valeurs

Deux applicateurs sont utilisés pour affecter (<-) ou ajouter (<=) une (ou plusieurs) valeurs à un attribut. Il s'agit là des applicateurs essentiels pour augmenter l'ensemble des informations dont dispose la base. L'adjonction de valeurs est contrainte par l'ensemble des filtres qui limitent leur évantail. Si une valeur indésirable souhaite être ajoutée, elle conduit à l'activation de réflexes d' "erreurs". Par exemple affecter la valeur -5 à un attribut qui ne supporte que les nombres positifs, entraîne l'activation du réflexe \$si-err-ajout, lequel peut être défini par l'utilisateur à sa convenance.

L'opération d'affectation ou d'ajout, si elle n'est pas stoppée et rejetée par les filtres, conduit aux déclenchements des réflexes \$si-ajout, \$regle-ajout et \$cont-ajout. Le premier contient une transmission, le second un ensemble de règles à activer, le troisième introduit une continuation dans le déroulement des opérations.

Voici l'ordre dans lequel MERING organise ces opérations afin de mener à bien ses tâches d'affectation et d'ajout. Soit V la valeur à adjoindre à l'attribut:

Opérations de filtrage

- 1) Si le domaine de l'attribut est différent de NIL de \$INDEF ou de \$VALEURS, voir si V correspond à ce domaine (ou à l'un des domaines s'il s'agit d'une liste). Si la valeur ne correspond pas, déclencher '\$si-err-ajout' et sortir.
- 2) Si le domaine de l'attribut est égal à NIL ou \$INDEF ou \$VALEURS, et si la facette '\$valeurs' est non vide, regarder si V correspond à l'une des valeurs de cette facette. si V ne correspond pas, déclencher le réflexe contenu dans '\$si-err-ajout' et sortir.
- 3) Si le contenu de la facette '\$req' est non vide, alors l'évaluer. Si le résultat de cette évaluation est NIL, alors déclencher le réflexe '\$si-err-ajout' et sortir.
- 4) Si le mode de l'attribut est \$UNI (contenu de la facette '\$mode'), et si V est une liste, ou s'il s'agit d'une opération d'ajout et que l'attribut possède déjà une valeur en propre, alors déclencher le réflexe '\$si-err-mode' et sortir.

Activation des réflexes

- 5) placer V dans la facette '<-' de l'attribut. Si l'applicateur incriminé est <- alors utiliser le message <-fac: '<-', si il s'agit de <=, employer<=fac: '<-.
- 6) Si le contenu de la facette '\$si-ajout' est non vide, alors évaluer cette transmission.
- 7) Si le contenu de la facette '\$regle-req' est non vide, alors déclencher chacune des règles qu'il contient.
- 8) si le contenu de la facette '\$cont-ajout' est non vide, alors court circuiter toutes les opérations en attente, retourner au 'toplevel' et évaluer le contenu de cette facette. (en d'autres termes, soit C le contenu de cette facette, effectuer (SEND: C)); sinon retourner l'entité en question comme résultat de cette opération d'affectation ou d'ajout.

Exemple:

Voici un exemple qui concerne un petit monde de robots, assez restreint du reste. IL exprime simplement qu'il existe un robot, qui s'appelle R2D2. A l'état initial, sa pince préhensible est vide et il se trouve dans son alcôve. Il est possible de le manipuler par l'intermédiaire d'actions: aller-a, prendre, déposer, etc...

En voici les principaux types (ceux qui ne sont pas définis explicitement dans cet exemple sont supposés être des sous-types de '\$objet').

```
(type => $objet robot
  lieu $dom lieu
    $mode $uni,
  pince $valeurs (vide pleine)
    $default vide,
  tient $dom chose
    $mode $uni
    $req (## < (ce max-poids) & ce pince = vide)
    $si-ajout (% pince <- pleine)
    $cont-ajout "c'est pris !"
    $si-err-ajout ("impossible de prendre " print & ##),
  max-poids $dom nombre
    $default 5,
)

(type => $objet action
  newrep methode: "yes sir !",
)

(type => action aller-a
  agent $dom robot
  origin $dom lieu
    $req (## = (ce agent lieu ?))
    $si-req (ce agent lieu ?),
  destin $dom lieu
    $si-ajout (ce agent lieu <- ##),
  newrep methode: () "yes sir !",
)

(type => action prendre
  agent $dom robot
```



```

    theme $dom chose
      $si-ajout (ce agent tient <- ##),
  )

; un meuble est défini à la fois comme un lieu et comme une chose
; cela signifie qu'un meuble peut être destination d'un déplacement
;
; et theme d'une préhension      ;
(type => (chose lieu) meuble
  poids $default 15,
  supporte $dom chose,
)

(type => meuble table)

(type => meuble chaise)

(lieu => alcove)

(table => tableA
  supporte <- cube,
)

(table => tableB)

(chose => cube
  poids $default 2,
)

(robot => R2D2)

```

On note immédiatement que ce robot, ne peut porter des objets de plus de 5 kilogs, et pas plus d'un à la fois. Cette exigence est placée dans la facette **\$req** de son attribut **tient**. De plus, le fait de prendre un objet, place la pince dans un état plein, comme l'indique le réflexe **\$si-ajout** de ce même attribut.

L'action **aller-a** a comme effet de bord de changer la position du robot. Cependant, elle suppose, pour s'accomplir, que l'origine du déplacement doit correspondre avec la position du robot.

Pour déplacer le robot, il suffit de lui donner l'ordre d'aller quelque part. Par exemple:

```
: (aller-a => ()
  agent <- R2D2
  destin <- tableA
)
= "yes sir !" ; résultat de l'application de;
               ; la méthode 'newrep' ;
```

Le résultat de cette opération est un nouvel état du micro-monde:

```
: (aller-a1 print & R2D2 print)
(ALLER-A => ALLER-A1
  AGENT <- (R2D2),
  ORIGIN <- (ALCOVE),
  DESTIN <- (TABLEA),
)

(ROBOT => R2D2
  LIEU <- (TABLEA)
)
= R2D2
```

On constate que tout s'est bien passé; à partir des informations fournies lors de la création du déplacement **aller-a1**, le système a mis à jour les valeurs des attributs **origin** de cette entité, ainsi que l'attribut **lieu** de **R2D2**, grâce au réflexe '\$si-ajout' placé dans l'attribut 'agent' de l'action 'aller-a'.

```
: (prendre => ()
  agent <- R2D2
  theme <- tablea
)
"impossible de prendre cet objet"
= TABLEA

: (prendre => ()
  agent <- R2D2
  theme <- cube
)
= "c'est pris !"
```

```

; la continuation placée sur l'attribut 'tient' ;
; court-circuite les évaluations en suspens,
; telles que le déclenchement de la méthode 'newrep';

: (r2d2 print)
  (ROBOT => R2D2
    LIEU   <- (TABLEA),
    TIEN   <- (CUBE),
    PINCE  <- (PLEINE),
  )
= R2D2 ; et tout s'est bien passé ;

```

Un exemple plus développé de ce genre est présenté dans chapitre III. Il concerne un petit univers de jeu d'aventure, montre comment il est possible de manoeuvrer aisément personnages dans un tel monde.

IV.2.2.4.3 Suppression de valeurs

L'applicateur **del** sert à supprimer une valeur dans un attribut. Son fonctionnement est beaucoup plus simple que dans le cas des autres applicateurs mentionnés ci dessus.

Soit **V** la valeur passée en argument de cette transmission si **V** appartient à l'ensemble des valeurs que possède en propre cet attribut, alors l'évaluation de cet applicateur revient à supprimer cette valeur, à déclencher le réflexe '\$si-suppr' et retourner l'entité receveuse comme résultat.

Si **V** est la valeur **NIL**, tout le contenu de la facette de l'attribut est effacé.

IV.2.2.5 Prédicats de correspondance

MERING dispose d'un certain nombres de comportements qui lui permettent de tester l'égalité, l'analogie ou la ressemblance, entre deux entités ou deux valeurs en général.

Dans les problèmes d'intelligence artificielle, l'égalité n'est pas un critère suffisant pour rendre compte des correspondances qui existent entre deux informations. Très souvent il ne s'agit pas de comparer deux entités mais de savoir si une entité a quelque chance de succès de correspondre à la description qui en est faite dans un attribut. Par exemple, est ce que 'toto' pourrait être considéré comme le propriétaire du bateau 'ivre'. Cela ne peut se faire qu'en comparant une virtualité, c'est à dire une définition caractérisée par un ensemble de contrainte, et une entité réelle.

Il est aussi important de savoir si un attribut contient une, ou plusieurs, entités d'un certain type, en tenant compte des mécanismes d'héritage. Par exemple si l'on sait que 'toto' possède des animaux familiers (Fido et Minou), on voudrait savoir, s'il possède un chien. Rien, dans l'attribut ne permet de dire directement oui ou non. Il faut examiner chacune des valeurs pour savoir si l'un de ces animaux est un chien.

De plus on désire que les prédicats employés témoignent d'une certaine souplesse. Qu'ils ne refusent pas d'exprimer leur avis parce que l'argument est une liste et non une entité, ou qu'il s'agit d'une simple s-expression.

Afin de répondre à tous ces souhaits nous avons incorporé dans MERING plusieurs comportement qui ont pour but de comparer des informations entre elles, de la manière la plus adéquate qui soit. Ces opérateurs sont de trois types: égalité, contenance et correspondance (nous parlons ici de correspondance lorsqu'il s'agit de comparer deux descriptions, en tenant compte aussi bien des valeurs réelles que des contraintes imposées).

Dans tous les cas, ces égalités ne font intervenir que la description des entités en tant que 'schéma' (c'est à dire définis comme un ensemble d'attributs) sans prendre en compte les méthodes ni les applicateurs.

IV.2.2.5.1 Les prédicats d'égalité

Ceux ci sont au nombre de deux:

= **methode:** (v) compare deux entités ensemble. Bien que le test qu'elle effectue suppose l'égalité absolue, celle ci n'est pas définie de la même manière pour toutes les entités receveuse. En effet ce prédicat dépend du type de cette entité.

1) s'il s'agit d'un nombre ou d'une chaîne de caractère, ce prédicat suppose l'égalité classique.

```
: (3 = 3)
= T
```

```
: (4 = 3)
= NIL
```

2) En revanche s'il s'agit d'une entité de type 'cell' (ou d'une séquence qui est un sous type de la cell), c'est avec son contenu que s'effectue la comparaison.

```
: (cell => a | <- 5)
= A ; construction de la cellule A qui contient la
valeur 5;
```

```
: (a = 5)
= T ; A reçoit le message d'égalité avec 5 comme
argument;
```

```
: (5 = a)
= T ; ici c'est 5 qui reçoit le message d'égalité
```

```

: (a = 3)
= NIL

: (seq => b | <- '(3 4 5))
= B ; construction d'une sequence ;

: (b = '(3 4 5))
= T ; comparaison dans un sens ;

: (b = '(4 5 3))
= T ; l'ordre des elements n'a pas d'importance;

: ('(4 3 5) = b)
= T ; ça marche dans l'autre sens aussi ;

: (b = '(3 4))
= NIL

: (b = '(4 5 6))
= NIL

```

3) Pour toutes les autres entités, la comparaison s'effectue attribut par attribut sur la base des valeurs que contiennent les attributs. De plus les deux entités doivent être issues d'un même type ou bien que le type de l'un soit un sous type de celui de l'autre.

```

: (type => $objet truc
  a $dom $indef,
)
= TRUC

: (type => truc machin
  b $dom personne,
)
= MACHIN

: (type => $objet chose
  a $dom $indef,
  b $dom personne,
)
= CHOSE

: (truc => | a <- bof)

```

```

= TRUC1

: (truc => | a <- plop)
= TRUC2

: (trucl = truc2)
= NIL
; Leurs attributs comportent des valeurs différentes ;

: (machin => machl | <- bof | = trucl)
= T
; machin est sous type de truc ;

: (machl b <- toto)
= MACH1

: (trucl = machl)
= NIL
; machl possède un attribut comportant une valeur en plus;

: (chose => chosel | a <- bof | b <- toto)
= CHOSE1

: (chosel = machl)
= NIL
; machin n'est pas un sous type de chose, ni l'inverse;

```

= **applic: (v)** : Cet applicateur reprend l'essentiel de la méthode précédente mais en comparant cette fois non pas deux entités ensemble, mais le contenu d'un attribut avec une entité (ou une liste d'entités ou simplement une s-expression) passée en argument.

```

: (toto age <- 40 )
= TOTO

: (toto age = 40)
= T ; c'est le contenu de 'age' qui est comparé;

: (toto passions <- '(musique lecture tennis))
= TOTO

: (toto passions = '(musique tennis lecture))
= T ; l'ordre n'a toujours pas d'importance ;

```

```
: (toto passions = tennis)
= NIL
```

IV.2.2.5.2 Les prédicats d'appartenance

Nous appelons 'prédicats d'appartenance' des applicateurs qui utilisent une relation d'appartenance dérivée de celle de la théorie des ensembles. Ils permettent de répondre aux questions du genre "cette entité contient elle une entité d'un certain type".

Répondre à une telle question nécessite de prendre en compte non seulement les valeurs des attributs, mais aussi les contraintes qui les caractérisent.

Dans les types 'cell' ou 'seq', ces comportements sont en fait des méthodes qui agissent directement au niveau de leur attribut 'cell'. Soit F l'applicateur en question, la méthode M qu'il introduit se définit de la manière suivante:

```
M methode: (v) (% 'cell F v),
```

cont?-un applic: (V) Si V est une représentant, cet applicateur retourne le premier élément qui soit égal (au sens de l'égalité '=' décrite plus haut) à V. En revanche, s'il s'agit d'un type, il ramène la première valeur de l'attribut en question qui soit issue du type V.

```
: (type => $objet personne
  age      $dom $nombre,
  taille   $dom $nombre,
  profession $dom profession,
  passions $dom activite,
)
= PERSONNE
```



```

: (type => activite sport)
= SPORT

: (type => activite art)
= ART

: (art => dessin & sport => voile & sport => ski)
= SKI

: (personne => jules
  passions :: (voile dessin ski),
  )
= JULES

: (jules passions cont?-un activite)
= VOILE

: (jules passions cont?-un sport)
= VOILE

: (jules passions cont?-un art)
= DESSIN

: (jules passions cont?-un loisir)
= NIL

```

cont?-des applic: (V) si V est un représentant, cet applicateur retourne tous les éléments qui sont égaux (au sens de l'égalité '=') à V. Sinon, si V est un type, il produit l'ensemble des valeurs de l'attribut qui soient issues de V.

```

: (jules passions cont?-des sport)
= (VOILE SKI)

: (jules passions cont?-des activite)
= (VOILE DESSIN SKI)

: (jules passions cont?-des art)
= DESSIN

: (jules passions cont?-des loisir)
= NIL

```

applic: (V) agit de manière semblable aux deux applicateurs précédents, mais prend en compte le fait

que V puisse être une liste.

- Si V est une entité, == se comporte de la même manière que **cont?-un**, et retourne le premier élément égal ou issu de V.

- Si V est une liste, alors il ramène le premier élément de l'attribut qui soit égal à l'un des représentants, ou issu de l'un des types que V contient. L'ordre des recherches dépend de la manière dont sont rangés les éléments dans V.

```
: (jules passions == (loisir voile art))  
= VOILE
```

```
: (jules passions == (loisir art voile))  
= DESSIN
```

```
: (jules passions == (machin sport art))  
= VOILE
```

IV.2.2.5.3 Les prédicats de filtrage

Ce groupe comporte un certain nombre de comportements qui s'attachent à des comparaisons plus larges, à des correspondances plus vagues que l'égalité ou l'appartenance, et prennent en compte des informations issues des contraintes que renferment les facettes des attributs.

Ces comportements permettent d'effectuer des filtrage (pattern-matching) sur les éléments de la base.

De plus, il est généralement possible de n'obtenir qu'un filtrage partiel, en déclarant la liste des attributs sur lesquels cette compararaison doit s'effectuer.

De plus, aucun de ces comportements ne déclenche de réflexes. Ils sont donc "sains" d'emploi.

=?= applic: (obj attr) compare deux attributs au niveau des facettes \$dom et \$valeurs. Si les facettes valeurs sont identiques, ou que tous les domaines de l'entité receveuse sont des sur-types des domaines de l'attribut passé en argument, ce comportement retourne T. Autrement il ramène NIL.

Reprenons notre type 'personne' défini à la section précédente, et créons un nouveau type, GUGUS:

```
(type => () gugus
  age      $dom $nombre,
  passions $dom sport,
)
= GUGUS

: (jules age =?= gugus taille)
= T ; ce sont deux nombres ;
    ; Notons que l'une des entit{ est ;
    ; un type l'autre un repr{sentant ;

: (jules passions =?= gugus passions)
= T ; le domaine des passions du type GUGUS;
    ; est issu du domaine des passions de ;
    ; jules ;

: (gugus passions =?= jules passions)
= NIL ; mais l'inverse n'est pas vrai ;
```

=?= methode: (obj lattr) compare deux entités sur la base d'une liste d'attributs passée en argument (lattr). Cette comparaison est effectuée attributs par attribut, en utilisant le même mécanisme que pour l'applicateur =?=. Cette méthode se décrit donc formellement comme suit:

```
=?= methode: (obj lattr) (testall X lattr
                          (% X =?= obj X)),
```

Ex:

```
: (jules =?= gugus '(age passions))
```

= T ; sur deux attributs seulement c'est bon;

: (jules =?= gugus)

= NIL ; mais pas sur tous les attributs ;

joint? **applic:** (**obj**) Cet applicateur indique si l'entité passée en argument est compatible avec les facettes **\$dom** et/ou **\$valeurs** de l'attribut, c'est à dire si 'obj' fait partie des éléments de la facette '**\$valeur**', ou bien si 'obj' est une classe (ou un représentant d'une classe) dont l'un des éléments de la facette '**\$dom**' est issu. Il est défini ainsi:

```
joint? applic: (obj) (or (% # fac? '$valeurs | == obj)
                        (if (obj type?)
                            (% # fac? '$dom | == obj)
                            (% # fac? '$dom | == (obj
is))))),
```

Ex:

: (jules age joint? 3)

= T ; oui puisque c'est un nombre ;

: (jules passions joint? ski)

= T ; oui puisque c'est un sport ;
; donc une activité ;

: (jules passions joint? 3)

= NIL ; non, ce n'est pas un nombre ;

joint?+ applic: (**obj**) est identique à l'applicateur précédent à ceci près qu'il effectue un test supplémentaire sur l'entité 'obj', en appliquant le filtre **\$req**. Attention, la facette **\$req** étant évaluée, il peut se produire des effets de bord indésirables.

joint? methode: (**obj**) signale s'il existe un lien entre deux entités, c'est à dire que l'une des alternatives est satisfaite:

1) l'entité 'obj' est valeur de l'un des attributs de

l'entité receveuse.

2) l'entité 'obj' est défaut de l'un des attributs de l'entité receveuse.

3) l'entité 'obj' est compatible (au sens de l'applicateur 'joint?') avec l'un des attributs de l'entité receveuse.

4) il existe une entité X telle que (% joint? X) est vrai et (X joint? obj) l'est également.

Le résultat est NIL lorsqu'il est impossible de faire le lien. Autrement, la méthode retourne une liste qui comprend l'ensemble des couples <entité,attribut> qui composent ce lien.

Ce comportement est très pratique pour voir s'il existe des liens entre entités.

```
: (sport => natation
  element $dom element-phys,
)
= NATATION

: (element => terre & element => eau)
= EAU

: (jules passions <= natation)
= JULES

: (jules joint? 3)
= ((jules age)) ; le lien s'effectue par l'age ;

: (jules joint? terre)
= ((jules passions)(natation element))
; le lien est valable puisqu'on ne connaît pas ;
; exactement la valeur de l'attribut element de ;
; natation ;

: (natation element <- eau)
= NATATION

: (jules joint? terre)
= NIL
; maintenant on sait qu'il s'agit d'eau ;
```

match? methode: (lobj lattr) compare l'entité receveuse avec la liste d'entité 'lobj' sur la base de comparaisons =?=. Cette méthode retourne la liste des entités qui correspondent. Sa définition est la suivante:

```
match? methode: (lobj lattr) (forall X lobj
                               (if (% =? X lattr) X)),
```

D'autres comportements et systèmes de filtrage ont été planifiés, mais n'ont pas encore été implantés dans le langage.

Il faut remarquer qu'il est facile de se créer des applicateurs de ce type. Par exemple pour créer un applicateur **filtre** qui va comparer deux entités domaine par domaine, puis si cela marche, valeur par valeur, il suffit d'écrire:

```
filtre methode: (obj) (forall X (% listattr)
                          (and (% X =? obj X)
                               (% X = obj X))),
```

Il est ensuite possible de réaliser toutes les variantes possibles, en utilisant différents applicateurs de test.

IV.2.2.6 Les manipulateurs de listes

La structure de donnée fondamentale est la liste (ou la séquence). Il est donc à la fois normal et indispensable de posséder plusieurs comportements ayant pour objet la manipulation de ces listes.

Dans tous les cas il s'agit de manipuler la facette '<-' de l'attribut, c'est à dire l'information que possède un attribut. C'est pourquoi tous ces comportements sont avant tout des applicateurs. Cependant, les types `seq` et `$seq` qui implémentent la structure de liste, disposent de méthodes dont l'action est identique à ces applicateurs, mais agissent implicitement au niveau de l'attribut 'cell'.

Ces comportements se divisent en trois groupes qui permettent de manipuler ces structures en les concevant selon trois modes différents: listes simples, listes indexées et "attributs liés".

Avec ce dernier mode il est possible de prendre en compte des relations d'associations entre éléments appartenant à deux attributs différents d'une même entité, afin de construire des structures analogues aux "a-listes" du langage LISP.

Tous ces comportements sont actifs, c'est à dire qu'ils déclenchent les réflexes associés aux facettes de l'attribut. En particulier les applicateurs de lecture déclenchent les réflexes '\$default', '\$si-lu', '\$si-req' et '\$regle-req' de façon identique à l'applicateur ? et ??. Ceux d'affectation et d'ajout emploient les mêmes facettes que <= et <-.

les listes simples

Il s'agit des comportements de base de manipulation des séquences, que celles ci soient associées à des attributs, ou qu'elles apparaissent de manière indépendante, c'est à dire comme des entités issues des types **seq** ou **\$seq**. Dans ce cas les applicateurs que nous décrivons ici deviennent des méthodes.

1? applic: () ramène le premier élément de la séquence.

```
: (personne => jules
  passions <- (voile dessin ski),
)
= JULES

: (jules passions 1?)
= VOILE
```

reste applic: () retourne la séquence privée de son premier élément.

```
: (jules passions reste)
= (DESSIN SKI)
```

der applic: () renvoie le dernier élément.

```
: (jules passions der)
= SKI
```

!! applic: () ramène le premier élément et le supprime de la séquence. Analogue au NEXTL de VLISP.

```
: (jules passions ?)
= (VOILE DESSIN SKI)

: (jules passions !!)
= VOILE

: (jules passions ?)
= (DESSIN SKI)
```


l<= applic: (V) insère l'argument V en tête de séquence (alors que l'applicateur <= le place à la fin) et ramène l'entité receveuse du message comme résultat. Semblable au NEWL de VLISP

```
: (jules passions ?)
= (VOILE DESSIN SKI)
```

```
: (jules passions l<= lecture | passions ?)
= (LECTURE VOILE DESSIN SKI)
```

Les listes indexées

Quelques applicateurs permettent de traiter les séquences comme s'il s'agissait de tableaux unidimensionnels.

n? applic () donne le nombre d'éléments que contient une séquence.

```
: (jules passions ?)
= (LECTURE VOILE DESSIN SKI)
```

```
: (jules passions n?)
= 4
```

I? applic: (I) ramène le $i^{\text{ème}}$ élément de la séquence. S'il n'appartenait au langage de base, l'applicateur I? pourrait être réécrit sous la forme:

```
I? applic: () (% # I?),
```

Ex:

```
: (jules passions I? 3)
= DESSIN
```

I<- applic: (I V) remplace le $i^{\text{ème}}$ élément de la

séquence par l'argument V.

```
: (jules passions I<= 2 equitation)
= JULES
: (jules passions ?)
= (LECTURE EQUITATION DESSIN SKI)
```

I<= applic: (I V) insère la valeur V avant le i^{ème} élément de la séquence.

```
: (jules passion I<= 2 natation | passions ?)
(LECTURE NATATION EQUITATION DESSIN SKI)
```

L'opérateur I<= pourrait donc se définir de la manière suivante:

```
I<= applic: (V) (% # I<= 1 V),
```

Les attributs liés

La structure des entités s'étale en trois niveaux: le niveau de l'entité lui-même, celui des attributs, enfin celui qui se rapporte aux facettes. Cette division en niveaux n'est pas toujours suffisante: il est bon de pouvoir relier des attributs entre eux, et ainsi créer des associations entre étiquettes et valeurs, sans passer par l'affectation d'une valeur à un attribut.

Cette construction, qui présente de nombreuses analogies avec les a-listes, fait partie de MERING. De plus, il est possible de caractériser cette structure à partir du langage lui-même. En d'autres termes, il s'agit là d'une des premières applications des facilités de MERING pour constituer des structures de représentations plus adéquates par rapport à une application donnée, ou seulement plus au goût du programmeur.

Créer une association entre attributs, revient à définir un attribut comme "attribut étiquette", qui portera les identificateurs nécessaires à la liaison, et un autre comme

"attribut valeur", qui contient les informations à mémoriser.

Par exemple: imaginons que nous voulions créer un type 'scenario', défini comme une liste d'évènements accomplis par des personnes. (Il serait possible de définir la notion de scénario de façon très différente).

```
: (type => $objet scenario
  roles      $dom $indef
             $assoc personnes,
  personnes  $dom personne,
  events     $dom $indef
             $assoc event-list,
  event-list $dom evenement,
)
= SCENARIO
```

L'attribut 'roles' contient l'ensemble des personnages qui tiennent un rôle, c'est à dire qui sont les actants du scénario. Par exemple dans le scénario célèbre du restaurant dû à SCHANK (Schank.81), ces personnages sont le client, le serveur, le patron du restaurant etc...

```
: (type => scenario aller-restaurant
  roles <- (client serveur patron),
  events <- (entrer-rest commander
            manger-repas payer sortir-rest),
)
= ALLER-RESTAURANT
```

Les facettes \$assoc servent à créer un lien entre les "attributs étiquettes" (roles et events) et les "attributs valeurs" (personne et event-list). Les premiers servent ici d'index pour répertorier des valeurs dans les seconds. Cette construction est intéressante car elle permet de dégager des noms généraux (client, serveur, commander, etc...) des entités réelles qui les représentent.

Par exemple si l'on sait que le client s'appelle "Paulo", le serveur "Fred", et le scénario de commande "scen12", il sera

possible de placer ces informations dans les attributs associés: 'personnes' dans le premier cas et 'event-list' dans le second. Une scène de restaurant, une fois connus les acteurs et les évènements prendra donc l'allure suivante:

```
: (aller-restaurant => rest1
  roles      <- (client serveur patron),
  personnes  <- (paulo fred bebert)
  events     <- (entrer-rest commander
                manger-repas payer sortir-rest),
  event-list <- (scen34 scen12 scen45 payer6 scen38),
)
= REST1
```

Nous n'entrerons pas ici dans des discussions faisant intervenir la structure et l'utilisation des scénarios. Nous allons seulement décrire l'ensemble des applicateurs qui permettent de manipuler ces attributs associés.

A? **applic: (sel)** est le premier d'entre eux, et certainement le plus important. Il permet de retrouver une information en sachant le nom de l'attribut étiquette, qui est l'attribut receveur du message, et celui de l'index passé en argument. Si le sélecteur est **all**, la liste des valeurs associées est retournée.

```
: (rest1 roles A? serveur)
= FRED

: (rest1 events A? commander)
= SCEN12

: (rest1 roles ?)
= (CLIENT SERVEUR PATRON)

: (rest1 roles A? all)
= (PAULO FRED BEBERT)
```

Ce comportement peut être défini en MERING de la manière suivante:

```
A? applic: (sel) (if (sel = all)
```

```
(% (% # assoc?) ?)
(% (% # assoc?) I? (% # N? sel))),
```

L'aplicateur **assoc?** a pour fonction de retourner le nom de l'attribut-valeur associé à l'attribut-étiquette.

On constate ici qu'il est possible de calculer le nom d'un attribut au cours d'une transmission.

A<- applic: (sel val) est l'aplicateur d'affectation d'une valeur dans un attribut associé. L'attribut receveur est toujours l'attribut-étiquette. Le nom du sélecteur ainsi que la valeur à placer, est passée en argument.

```
: (rest1 roles A<- serveur pierre)
= REST1
```

```
: (rest1 roles A? all)
= (PAULO PIERRE BEBERT)
```

Ce comportement se définit ainsi:

```
A<- applic: (sel val) (% (% # assoc?) I<- (% # N? sel) val),
```

CONCLUSION

Nous avons présenté dans cette étude, un langage d'acteur destiné à représenter et à manipuler des connaissances. Nous avons voulu montrer qu'un langage de représentation doit être aussi un langage de programmation, pour que soit offert, au programmeur en Intelligence Artificielle, un environnement de programmation aussi efficace que possible.

Il s'agissait aussi d'appréhender la représentation des connaissances sous un formalisme "objet" tout en montrant les capacités de flexibilité, de généralité et modularité d'une telle approche.

MERING est un langage ouvert. Dans le chapitre III nous avons montré comment il est facile d'augmenter les capacités du langage pour l'adapter à un domaine d'application spécifique. MERING est un langage particulièrement adéquat pour servir de base d'implantation à des systèmes experts de la "nouvelle génération" c'est à dire à des programmes qui ne peuvent se satisfaire d'un seul mécanisme de contrôle ou de représentation: frames ou règles de production. De fait, l'intégration de ces deux modes de représentation au sein d'un même langage rend l'architecture du système beaucoup plus cohérente.

En outre, nous avons souligné l'importance des applicateurs et des réflexes: ils permettent de concevoir les attributs sous un regard neuf en les considérant comme des entités locales qui

servent de relais entre l'entité globale et ses valeurs propres. Les réflexes représentent alors les comportements spécifiques aux attributs.

De plus, nous avons cherché à dépasser le formalisme des 'langages orientés objet' tels que SMALLTALK, à casser le cadre étroit qu'ils imposent entre types d'une part et représentants d'autre part, pour introduire des orientations qui nous apparaissent comme indispensables à la représentation des connaissances: notion de contraintes et de réflexes, possibilité d'accorder aux types un statut d'entité à part entière, sans devoir recourir à des mécanismes artificiels comme ceux des méta-classes en SMALLTALK-80, etc...

Le système ne prétend pas actuellement résoudre les problèmes spécifiques à la conceptualisation. Il doit être considéré comme un outil de développement souple, pratique et puissant pour la réalisation d'applications en Intelligence Artificielle.

De nombreux points du langage (comme les classes et les règles de production) de même que les possibilités offertes par l'utilisation de contraintes, de continuations et de recouvrement d'erreurs (mais peut on encore parler d'erreurs lorsqu'il s'agit d'un fonctionnement normal du système) n'ont été qu'à peine ébauchés: des études ultérieures permettront à la fois d'en mieux comprendre les fondements théoriques et d'en dégager les conséquences pratiques.

Nous envisageons de développer MERING selon quatre axes:

1) Réécrire l'interprète actuel de manière à le rendre plus efficace, soit en le compilant, soit en l'écrivant dans un autre langage (C par exemple).

2) Enrichir l'environnement du langage, en créant de nouveaux utilitaires de programmation (trace, superviseur, sondes,...) d'une part, et des éléments de représentation adaptés aux recherches actuelles (scénarios, représentations temporelles et spatiales, intégration de plusieurs traitements logiques...) d'autre part.

3) Développer des applications plus conséquentes que les exemples qui ont été fournis: analyseurs de langage naturel, programmes de dialogues, systèmes experts, etc...

4) Etendre et généraliser les principes mêmes du langage, en lui incorporant de nouvelles possibilités: co-routines, mécanismes de filtrage plus élaborés, extension de la notion d'acteur, etc...

ANNEXE A

```

*****
;
;
;
Liste des principales entites utilisees par MERING
;
;
$sysroot      ;
$sysobj       ;
$objet        ;
cell          ;
seq           ;
$cell         ;
$seq          ;
ws            ;
classe        ;
wsystem       ;
;
*****
;

```

```

base de tout ;
'$SYSROOT
'($SYSROOT
( $TYPE ( <- $CLASS))
( <= ($APPLIC (V) (AFFECT 'FADD '% # V) ))
( ?? ($APPLIC () (FLIREO '% #)))
( /? ($APPLIC () (FLIRE '% # )))
( DEL ($APPLIC (V) (IF (NULL V)
(FACDEL '% # '<- )
(FREMOVE '% # '<- V)
& (MEVAL (FGET-I '% # '$SI-SUPPR))
& '%)))
( PRINT ($METHODE () (FPRINT '%)))
( ALL ($METHODE () (FGET-I '% '$INST '<-)))
( <- ($APPLIC (V) (AFFECT 'FMET '% # V)))
( IS ($METHODE () (FGET % '$ISA '<- )) )
(ISA? ($METHODE (V) (MEMQ V (FGETCLASSES '%))))
(/:= ($METHODE (V) ((VAR => %) := V) ))

```

```
(VAR? ($METHODE (NIL)))
(REP? ($METHODE (PINST '%)))
(TYPE? ($METHODE (PCCLASS '%)))
(=> ($METHODE (LST) (EVAL [ 'DINST % . LST] ))
($TYPE FEXPR))
(LSTMETH ($METHODE () (LSTMETHODE '%)))
(LISTATTR ($METHODE () (LSTATTR '%)))
(LSTAPPL ($METHODE () (LSTAPPL '%)))
)
'FRAME ))))))))
```

(\$classe \$sysroot) ; sert a bootstrapper le syst)me ;

; L'entite importante ;

(TYPE => \$SYSROOT \$SYSOBJ

; Le slot 'SLOTINI' contient toutes les valeurs par default des facettes ;

; ce slot est automatiquement chaine a tous les autres ;

```
SSLOTINI $DOM $INDEF
$DEFAULT NIL
$NUM $INDEF
$REQ NIL
$VALEURS NIL
$MODE $MULT
$SI-REQ NIL
$SI-LU NIL
$SI-AJOUT NIL
$SI-SUPPR NIL
$REGLE-RE NIL
$REGLE-LU NIL
$REGLE-AJOUT NIL
$SI-ERR-AJOUT (print "** erreur ajout valeur dans : " % #
& topmrg)
$SI-ERR-MODE (PRINT "** erreur ajout valeur univaluee dans : " % #
& topmrg)
$CONT-AJOUT NIL
$CONT-LECT NIL
$ORDRE-LECT ($default $sireq)
$QUESTION (quelle est la valeur du # de %),
```

; L'attribut de trace ;

```
$TRACE $DOM $BOOL
$DEFAULT NIL,
```

; Quelques methodes generales ;

```

FIRSTINST METHODE: () (CAR (% ALL)),
LASTINST METHODE: () (LAST (% ALL)),
NEWTYPE METHODE: () ('%),
NEWREP METHODE: () (FORALL X (% LSTATTR)
    (IF (FGET-I % X '<-)
        (MEVAL (% X FAC? '$SI-AJOUT))))),
QUESTIONNER APPLIC: () (PRINT (% # FAC? '$QUESTION)
    & % # <- (READ) ),
ISALL METHODE: () (FGETCLASSES '%),
ALLINST METHODE: () (ALLINST %),
ALLENT METHODE: () (ALLENT %),
HIERARCH METHODE: () (HIERARCH %);

; Tests de types
REP? APPLIC: () (AND (% # ?)
    (TESTALL X (% # FAC? '<-) (PINST X))),
TYPE? APPLIC: () (AND (% # ?)
    (TESTALL X (% # FAC? '<-) (PCALL X))),
NOMBRE? METHODE: () (NUMBP '%),
NOMBRE? APPLIC: () (OR (NUMBP (% # ?))
    (TESTALL X (FGET '% # '<-)(NUMBP X))
    (MEMQ '$NOMBRE (FGET '% # '$DOM))),
CHAINE? METHODE: () (STRINGP '%),
CHAINE? APPLIC: () (OR (STRINGP (% # ?))
    (TESTALL X (FGET '% # '<-)(STRINGP X))
    (MEMQ '$CHAINE (FGET '% # '$DOM))),
INDEF? METHODE: () (NIL),
INDEF? APPLIC: () (LET ((L (% # ?)))
    (OR (AND L (MAPCT L (LAMBDA(X) (NOT (PFRAME X))))))
        (AND (NULL L)
            (MAPCT (FGET-I % # '$DOM)
                (LAMBDA (X) (MEMQ X '($INDEF NIL $VALEUR)))))),
ENTITE? METHODE: () (T),
ENTITE? APPLIC: () (NOT (% # INDEF?)),
NUL? APPLIC: () (NULL (% # ?)),

; Manipulation de facettes
<=FAC: APPLIC: (F V) (FSET '% # F V),
<=FAC| APPLIC| (LST) (FSET '% ## (CAR LST) (CDR LST)),
<=FAC APPLIC: (F V) (FADD '% ## F V),
FAC? APPLIC: (F) (VALDE (FGET-I '% ## F)),
FACDEL APPLIC: (F V) (IF (NULL V)
    (FACDEL '% # F)
    (FREMOVE '% # F V),
FAC?? APPLIC: (F) (FGET-I '% ## F),
<=DEF APPLIC: ( V) (FPUT % # '$DEFAULT V) ,
<=DOM APPLIC: ( V) (FPUT % # '$DOM V) ,
<=DOM APPLIC: ( V) (FSET % # '$DOM V) ,

```

```

<-DEF APPLIC: ( V ) (FSET % # '$DEFAULT V) ,
<-SI-AJOUT APPLIC| (L) (FSET % # '$SI-AJOUT L),
<-SI-REQ APPLIC| (V) (FSET % # '$SI-REQ V),
<-APPLIC APPLIC| (V) (FSET % # '$APPLIC V) ,
<-METH APPLIC| (V) (FSET % # '$METHODE V) ,

```

```

; Manipulateurs de listes ;
COUNT APPLIC: ( ) (LENGTH (% # ?) ),
I? APPLIC: ( ) (CAR (% # ?) ),
RESTE APPLIC: ( ) (CDR (% # ?) ),
DER APPLIC: ( ) (CAR (LAST (% # ?) )),
I! APPLIC: ( ) (PROG1 (% # I?)(% # <- (CDR (% # ?) )),
I<= APPLIC: ( V)(% # <- (CONS V (% # ?) )),
I? APPLIC: ( V) (CAR (NTH V (% # ?) )),
I<- APPLIC: ( I V) (% # <- (MAKENTH I (% # ?) V) ),
I<= APPLIC: ( I V) (% # <- (INSERTH I (% # ?) V) ),
ASSOC? APPLIC: ( ) (CAR (FGET-I % # '$ASSOC)),
N? APPLIC: (V) (ITEM V (FGET-I % # '$<-)),
A? APPLIC: (SEL) (% (% # ASSOC?) I? (% # N? SEL)) ,
A<- APPLIC: (SEL VAL) (% (% # ASSOC?) I<- (% # N? SEL) VAL) ,
A<=1 APPLIC: (V) ((% # <= (CAR V))(% # ASSOC?) <= (CADR V)) ,
A<=N APPLIC: (LST)(CAR (MAPCAR LST '(LAMBDA (X) (% # A<=1 X)))) ,

```

```

; Quelques operations arithmetiques ;
/+ APPLIC: (V) (+ V (% # ?)),
/* APPLIC: (V) (* V (% # ?)),
/- APPLIC: (V) (- (% # ?) V),
// APPLIC: (V) (/ (% # ?) V),
/< APPLIC: (V) (< (% # ?) V),
/> APPLIC: (V) (> (% # ?) V),
INCR APPLIC: ( ) (% # <- (ADD1 (% # ?))),
DECR APPLIC: ( ) (% # <- (SUB1 (% # ?)))

```

```

; Predicats de correspondance ;
/= METHODE: (V) (AND
  (PFRAME V)
  (AND
    (OR (MEMQ V (FGETCLASSES '%))
      (MEMQ '% (FGETCLASSES V)))
    (LET ((L1 (% LSTATTR))
          (L2 (V LSTATTR)))
      (AND (EQUIV L1 L2)
            (TESTALL X L1
              (EQUAL (% X ?)(V X ?)))))),
/= APPLIC: (V) (EQUAL (% # ?) V),
CONT? APPLIC: (V) (IF (MEMBER V (% # ?)) T NIL),
CONT?-UN APPLIC: (V) (IF (PINST V)
  (EXIST X (% # ?)(IF (V = X) X))

```

```

                                (EXIST X (% # ?)(IF (X ISA? V) X))),
CONT?-DES APPLIC: (V) (IF PINST V)
                                (FORALL X (% # ?)(IF (V = X) X))
                                (FORALL X (% # ?)(IF (X ISA? V) X))),
==    APPLIC: (V) (IF (ATOM V)
                                (% # CONT?-UN V)
                                (EXIST X V
                                (% # CONT?-UN X))),
=?=    APPLIC: (OBJ ATTR) (IF
                                (OR
                                (LET ((L (FGET-I '% # '$VALEURS)))
                                (AND L
                                (EQUIV L (FGET-I OBJ ATTR '$VALEURS)))
                                (EXIST X (FGET-I OBJ ATTR '$DOM)
                                (X ISA? (FGET '% # '$ISA)))) T),
=?=    METHODE: (OBJ LATTR) (TESTALL X LATTR
                                (% X =?= OBJ X)),
JOINT? APPLIC: (OBJ) (MLET ((S SEQ))
                                (S <- (FGET '% # '$VALEURS))
                                (OR (S == OBJ)
                                (PROGN
                                (S <- (FGET '% # '$DOM)
                                (IF (PCLASS OBJ)
                                (S == OBJ)
                                (S == (OBJ IS)))))),
JOINT+? APPLIC: (OBJ) (AND (% # JOINT? OBJ)
                                (MEVAL (SUBST OBJ '## (FGET-I '% # '$REQ)))),
JOINT? METHODE: (OBJ) (ISLINK '% OBJ),
<=> APPLIC: (E S) (DYN-LINK '% # E S),
TELQUE: METHODE: (LST) (MEVAL
                                ['FORALL 'XX ['% ALLENT]
                                ['IF LST 'XX ]] )

                                $TYPE FEXPR ,
<-- METHODE: (V) (RECOPIE V '%),
=S> METHODE: (V) (FCLAMP '% V #),
KILL METHODE: () (IF (% PINST)
                                (FORALL X (% IS)
                                (X $INST DEL '%))
                                (FORALL X (% IS)
                                (X $SUB DEL '%))
                                & KILL '% ) ,
MATCH METHODE: (LOBJ LATTR) (FORALL X (% LATTR)
                                (IF (% =?= X LATTR) X)),
))))

```

Le type generique par default ;

```
(TYPE => $$SYSOBJ $OBJET
  TOUS: METHODE| (LST) (FORALL X [% ALLINST]
    (MEVAL [X . LST] )),
)
```

; La cellule ;

```
(TYPE => $$SYSOBJ CELL
  CELL $DOM $OBJET ,
  ? METHODE: () (FLIRE '% 'CELL ) ,
  <= METHODE: (V) (PROGN (SETQ % V)(FPUT '% 'CELL V) ) ,
  DEL METHODE: (V) (REMOVE '% 'CELL '<- V) ,
  <- METHODE: (V) (IF (LISTP V) ((SEQ => %) <- V)
    (FSET '% CELL '<- (SETQ % V))),
  NOMBRE? METHODE: () (NUMBP (% ?)),
  == METHODE: (V) (% CELL == V),
  /= METHODE: (V) (EQUAL (% ?) V),
  /< METHODE: (V) (< (% ?) V),
  /> METHODE: (V) (> (% ?) V),
  INCR METHODE: () (% <- (ADD1 (% ?))),
  DECR METHODE: () (% <- (SUB1 (% ?))),
  /+ METHODE: (V) (LET ((S (% ?)))
    (IF (AND (NUMBP S)(NUMBP V))
      (PLUS S V)
      NIL)),
  /- METHODE: (V) (LET ((S (% ?)))
    (IF (AND (NUMBP S)(NUMBP V))
      (DIFFER S V)
      NIL)),
  /* METHODE: (V) (LET ((S (% ?)))
    (IF (AND (NUMBP S)(NUMBP V))
      (TIMES S V)
      NIL)),
  ))))
```

; la structure de donnee: la sequence ;

```
(TYPE => CELL SEQ
  <- METHODE: (V) (FSET '% 'CELL '<- (SETQ % V)),
  <= METHODE: (V) (PROGN
    (FPUT '% 'CELL V)
    (SETQ % (% ?) '% ) ,
  COUNT METHODE: () (LENGTH (% ?)) ,
  1? METHODE: () (CAR (% ?)) ,
  LAST METHODE: () (CAR (LAST (% ?) )),
```

```

1! METHODE: () (PROG1 (% 1?)(% <- (CDR (% ? ) ) ),
1<= METHODE: (V) (% <- (CONS V (% ? ) ) ),
I? APPLIC: (V) (CAR (NTH # (% ? ) ) ),
I<- APPLIC: (V) (% <- (MAKENTH # (% ? ) V ) ),
I<= APPLIC: (V) (% <- (INSERTH # (% ? ) V ) ),
TOUS: METHODE| (LST) (FORALL X (% ? )
                (MEVAL [X . LST])),
))))

```

```

(TYPE => SEQ $SEQ)
(TYPE => CELL $CELL)

```

; les 'workspaces' pour les interactions avec le monde des fichiers ;

```

(TYPE => SEQ WS
  LOAD METHODE: (V)
    (LET ((F (IF V V %)))
      (FILE READ V)
      (WITH (EOF () (TERPRI)(INPUT) (&EOF))
        (ESLOOP &EOF
          (% <= (MEVAL (READ)))))
      V ),
  SAVE METHODE: ()
    (LET ((FILOUT ['DSK [% . 'VLI] () 055 ] ))
      (OUTPUT FILOUT)
      (FOR: (% ?) PRINT)
      (OUTPUT)
      FILOUT ),
)

```

; Creation d'un ws qui contient les classes systemes ;
 ((WS => WSYSTEM) <- (TYPE ALL))

ANNEXE B

```

*****
;
;
Les regles de production en MERING
;
Les classes definies sont:
;
REGLE
;
PAQUET
;
*****
;

```

```

nvlist0 (lx ly)
  (ifn (cdr ly)
    (append1 lx (car ly))
    (append1 lx
      (nvlist0 (car ly) (cdr ly)))))))))

```

```

nvlist (l)
(cond ((null l ) nil)
      ((null (cdr l)) (car l))
      (t (nvlist0 (car l)(cdr l)))))))))

```

```

gen-fonct (lv lq ld)
[(if (eq lq 'e) 'exist 'forall)
 lv
 [(if ld ld '$objet) 'allent]])))))

```

```

comprul(crul)
let ((lsoit (fget crul 'soit '::))
      (lquant (fget crul 'quant '::))
      (ldom (fget crul 'dom '::))
      (s ))
(setq s (mapcar lsoit '(lambda (x)
                        (gen-fonct x (nextl lquant)
                                      (nextl ldom))))))
(nvlist (append1 s ['if (fget crul 'action '::)

```

```
(fget crul 'condit ':'::) ])))))
```

```
(de domaine-de (obj )
```

```
(nth
```

```
(item obj (fget crul 'soit '<-))
```

```
(fget crul 'dom '<-))))
```

```
)
```

```
(type => $sysobj regle
```

```
args $dom $valeur,
```

```
soit $dom $valeur,
```

```
quant $dom $valeur,
```

```
dom $dom $objet,
```

```
condit $dom $trans,
```

```
action $dom $trans,
```

```
$trace $dom $bool
```

```
$default nil,
```

```
trace methode: (% $trace <- T),
```

```
detrace methode: (% $trace <- nil),
```

```
$regle-attr $dom $trans
```

```
$si-req (comprul '%),
```

```
execall: methode (larg) (forall arg larg (% exec: arg)),
```

```
exetype: methode (typ) (forall arg (typ allinst) (% exec: arg)),
```

```
exec: methode: (V) (progn
```

```
(if (ce $trace) (print " essai de la regle " %))
```

```
(if (meval (subst V (% args ?)) (% $regle-attr ? )))
```

```
(if (ce $trace)
```

```
(print " application de la regle " %))),
```

```
comp-gen methode: () (% regle <- (comprul '%)
```

```
& comp-decl-fwd '%
```

```
& comp-decl-bwd '% ),
```

```
)
```

```
(type => seq paquet
```

```
boucle $dom $bool,
```

```
toutes $dom $bool,
```

```
repete $dom $nombre
```

```
$default 10,
```

```
trace methode: () (forall X (% ?) (X $trace <- T)
```

```
& trace <- T),
```

```
detrace methode: () (forall X (% ?) (X $trace <- nil)
```

```
& trace <- nil),
```

```
trace methode: () (% $trace ?),
```

```
execall: (larg) (forall X larg (% exec: X)),
```

```
exetype: (typ) (forall X (typ allinst) (% exec: X)),
```

```

exec: methode: (-var-) (if (toutes = nil)
                          (if (boucle = nil)
                              (% exec1: -var-)
                              (% exec2: -var-))
                          (if (boucle = nil)
                              (% exec3: -var-)
                              (% exec4: -var-)))
exec1: methode: (-arg-) (exist -R- (% ?)
                        (-R- exec: -arg-)),
exec2: methode: (-arg-) (mlet ((C cell))
                          (ce continue <- nil)
                          (C <- 1)
                          (while (and (ce continue)
                                       (ce repete > @C))
                                (if (exist -R- (% ?)
                                      (-R- exec: -arg-))
                                    (ce continue <- T))
                                (C incr))),
exec3: methode: (-arg-) (forall -R- (% ?)
                        (-R- exec: -arg-)),
exec4: methode: (-arg-) (mlet ((C cell))
                          (ce continue <- nil)
                          (C <- 1)
                          (while (and (ce continue)
                                       (ce repete > @C))
                                (if (forall -R- (% ?)
                                      (-R- exec: -arg-))
                                    (ce continue <- T))
                                (C incr))),

```


Références bibliographiques

- (Aikins.79) Aikins J.S.
"Prototypes and production rules: an approach to knowledge representation for hypothesis formation"
Proc. IJCAI 79.
- (Berthelin.84) Berthelin J.B & Rady M.
"Déterminisme et contestation sont dans un bateau"
Colloque ARC 84 (à paraître)
- (Birtwistle.73) Birtwistle G.M., Dahl O.J, Myhraug B. & Nygaard K.
"SIMULA Begin"
Auerbach Publishers Inc. (1973)
- (Bobrow.77-a) Bobrow D.G., Winograd T. & al.
"GUS, a frame driven dialog system"
Artificial Intelligence n° 8 vol. 2 (1977)
- (Bobrow.77-b) Bobrow D.G & Winograd T.
"An overview of KRL, a knowledge representation language"
Cognitive Science n°1, Vol. n°1 (1977)
- (Bobrow.82) Bobrow D.G. & Stefik M.
"The LOOPS manual"
Xerox memo VLSI-81-13 (1982)
- (Brachman.77) Brachman R.J.
"A structural paradigm for representing knowledge"
Ph.D Thesis Harvard Univ (1977)

- (Brachman.79) Brachman R.J.
"On the epistemological status of semantic networks"
in "Associative networks: representation and use of
knowledge by computers"
Academic Press (1979)
- (Bundy.79) Bundy A., Byrd L. & Mellish C.
"Special purpose, but domain independent, inference mechanisms"
Proc. IJCAI 79 (?)
- (Briot.83) Briot J.P.
"L'instanciation dans les langages objets"
Journées Bigre sur les Langages Orientés Objets (1983)
- (Borning.82) Borning A. & Ingalls D.
"Multiple inheritance in SMALLTALK-80"
Proc. AAAI-82 (1982)
- (Carnap.50) Carnap R.
"Semantics and the philosophy of language"
Univ. of Chicago Press (1950)
- (Chailloux.78) Chailloux J.
"VLISP 10.3 Manuel de référence"
RT.76.78 Université Paris-8 (1978)
- (Charniak.76) Charniak E. & Wilks Y. Eds.
"Computational semantics"
North-Holland (1976)
- (Charniak.77) Charniak E.
"Ms Malaprop, a language comprehension program"
Proc. IJCAI 5 (1977)
- (Charniak.80) Charniak E., Riesbeck K. & McDermott D.
"Artificial Intelligence Programming"
Lawrence Erlbaum Ass. (1980)

- (Chouraqui.81) Chouraqui E.
"Contribution à l'étude théorique de la
représentation des connaissances.
Le système symbolique ARCHES"
Thèse d'état. Institut national polytechnique
de Lorraine. (1981)
- (Cointe.81) Cointe P.
"Fermeture dans les lambda interprètes. Application
aux langages LISP, SMALLTALK et PLASMA"
Thèse de 3ème cycle
Université Paris 6 (1981)
- (Cointe.81) Cointe P.
"VLISP, un langage objet ?"
Actes de la dixième école de printemps
Barèges (1982)
- (Durieux.81) Durieux J.L.
"Sémantique des liaisons nom-valeurs: application
à l'implémentation des lambda-langages"
Thèse d'état. Univ. P.Sabatier (1981)
- (Fahlman.79) Fahlman S.E.
"NETL: a system for representing and using real-world
knowledge."
MIT Press (1979)
- (Fargues.83) Fargues J.
"Contribution à l'étude du raisonnement, application
à la médecine d'urgence"
Thèse d'état Univ. Paris 6 (1983)
- (Ferber.83) Ferber J.
"Les langages objets dans la représentation des connaissances:
application au langage MERING"
Journée Bigre consacrée au langages orientés objets (1983)
- (Fox.79) Fox M.S.
"On inheritance in knowledge representation"
Proc IJCAI 79 (1979)

- (Frege.91) Frege G.
"Ecrits logiques et philosophiques " (1891)
Le Seuil (1971)
- (Goldberg.83) Goldberg & Robson
"SMALLTALK-80, the language and its implementation"
Addison Wesley (1983)
- (Goldstein.77) Goldstein I.P. & Roberts R.B.
"NUDGE, a knowledge-based scheduling program"
Proc. IJCAI 77
- (Greussay.77) Greussay P.
"Contribution à la définition interprétative
et à l'implémentation des lambda-langages"
Thèse d'état. Univ Paris 7 (1978)
- (Greussay.78) Greussay P.
"Le système VLISP 16"
Ecole polytechnique (1978)
- (Grau.83) Grau B.
"Analyse et représentation d'un texte d'après
le thème du discours"
Thèse de 3eme cycle. Univ. Paris 6 (1983)
- (Hewitt.71) Hewitt C.
"PLANNER, a language for proving theorems
and manipulating models in a robot"
Ph.D MIT (1971)
- (Hewitt.77) Hewitt C.
"Viewing control structure as patterns
of message passing"
Artificial Intelligence n° 3 Vol. 8 (1977)
- (Hewitt.80) Hewitt C. & Baker H.
"Actors and continuous functionals"

MIT memo (1980)

- (Kay.76) Kay A. & Goldberg A.
"SMALLTALK-72 instruction manual"
Xerox research center SSL-76-6 (1976)
- (Konolige.79) Konolige
"An inference net compiler for the PROSPECTOR
rule-based consultation"
Proc. IJCAI 79 (1979)
- (Laurière.82) Laurière J.L.
"Applicabilité industrielle des systèmes experts"
Actes des quatrièmes journées francophones
sur l'informatique. (1982)
- (Lehnert.83) Lehnert W. & al.
"BORIS - An experiment in in-depth understanding
of narratives"
Artificial Intelligence 20 (1983)
- (Lenat.75) Lenat D.
"BEINGS: Knowledge as interacting experts"
Proc. IJCAI 75
- (Lenat.82) Lenat D.
"AM, discovery in mathematics and heuristic search"
in "Knowledge based systems in Artificial Intelligence"
Mc Graw Hill (1982)
- (Liebermann.81) Liebermann
"A preview of ACT1"
MIT memo n°625 (1981)
- (Marcoux.79) Marcoux & Pomian
"PLASMA, un langage pour l'Intelligence Artificielle"
Congrès AFCET IAFC (1979)
- (Minsky.75) Minsky M.

"A framework for representing knowledge"
in "The psychology of computer vision"
Mc Graw Hill (1975)

(Novak.82) Novak S.

"GLISP users manual"
Stanford Univ. Report CS-82-895 (1982)

(Papert.76) Papert S. & Goldstein I.

"Artificial Intelligence, language &
the study of knowledge"
MIT memo (1976)

(Piaget.45) Piaget J.

"La formation du symbole chez l'enfant"
Delachaux & Niestle (1945)

(Piaget.76) Piaget J.

"Le comportement, moteur de l'évolution"
Gallimard (1976)

(Pomian.80) Pomian

"Contribution à la définition et à
l'implémentaion d'un interprète PLASMA"
Thèse de 3ème cycle
Universite Paul Sabatier - Toulouse (1980)

(Quillian.68) Quillian R.

"Semantic Memory"
in "Semantic Information Processing"
MIT Press (1968)

(Rady.83) Rady M.

"L'ambiguité du langage est elle la source
du non déterminisme des procédures de traitement"
Thèse d'état Paris 6 (1983)

(Raphael.68) Raphael B.

"SIR: Semantic information retrieval"
in "Semantic information processing"

MIT Press (1968)

- (Reiter.80) Reiter R.
"A logic for default reasoning"
Artificial Intelligence
Vol 13 n°1,2 (1980)
- (Reiter.83) Reiter R & Criscuolo G.
"Some representational issues in default reasoning"
Computers & Mathematics n°1 vol 9 (1983)
- (Roussel.75) Roussel
"PROLOG: manuel de référence et d'utilisation"
Université Aix-Marseille (1975)
- (Roberts.77) Roberts & Goldstein
"The FRL manual"
MIT memo n°409 (1977)
- (Rosenberg.80) Rosenberg St. T.
"Frame based text processing"
in "Frames conceptions and text undersxtanding"
De Gruyter (1980)
- (Sabah.78) Sabah G.
"Contribution à la compréhension
effective d'un récit"
Thèse de doctorat d'état paris 6 (1978)
- (Schank.77) Schank R. & Abelson R..
"Scripts, PLans, Goals and understanding"
Lawrance Erlbaum Associates (1977)
- (Schank.81) Schank R. & Riesbeck C.
"Inside computer understanding:
five programs plus miniatures"
Lawrance Erlbaum Ass. (1981)

- (Shortliffe.76) Shortliffe E.
"Computer based medical consultations, MYCIN"
American Elsevier (1976)
- (Siklossi.73) Siklossi L. & Dreussi J.
"An efficient robot planner which generates
its own procedures"
Proc IJCAI 73 (1973)
- (Smith.83) Smith R.G.
"STROBE: support for structured object
knowledge representation"
Proc. IJCAI 83 (1983)
- (Steele.78) Steele G. & Sussman G.
"Constraints"
MIT memo n° 502 (1978)
- (Steele.80) Steele G.
"The definition and implementation of a computer
programming language based on constraint"
Ph D. MIT. (1980)
- (Steels.79) Steels l.
"Procedural attachment"
MIT memo n° 543 (1979)
- (Stefik.79) Stefik M.
"An examination of a frame structured
representation system"
Proc IJCAI 79 (1979)
- (Stefik.82) Stefik M., Bell A.G., Bobrow D.G.
"Rule-oriented programming in LOOPS"
Xerox center memo n° VLSI-82-22 (1982)
- (Sridharan.78) Sridharan N. & Schmidt C.
"Knowledge directed inference in BELIEVER"
in "Pattern directed inference systems"
Academic Press (1978)

- (Thom.80) Thom.R
"Modèles mathématiques de la morphogénèse"
C.Bourgeois (1980)
- (Melle.81) Melle W., Scott A., Bennett J. & Peairs M.
"The EMYCIN manual"
Standford university"
Report n° 885 (1981)
- (Whitehead.1910) Whitehead N.R. & Russell B.
"Principia mathematica"
Cambridge Univ. Press (1910)
- (Winograd.72) Winograd T.
"Understanding natural language"
Academic Press (1972)
- (Winograd.83) Winograd T.
"Language as a cognitive process
Vol 1 : Syntaxe"
Addison-Wesley (1983)
- (Winston.81) Winston P.H. & Horn B.
"LISP"
Addison Wesley (1981)
- (Wittgenstein.1921) Wittgenstein L.
"Tractatus logico philosophicus" (1921)
Trad. Routledge & Keagan Paul (1963)
- (Woods.75) Woods W.
"What's in a link: foundations for
semantic networks"
in "Representation and undstanding"
Academic Press (1975)
- (Yonezawa.79) Yonezawa & Hewitt C.
"Modelling distributed systems"
Machine Intelligence n° 9 (1979)

